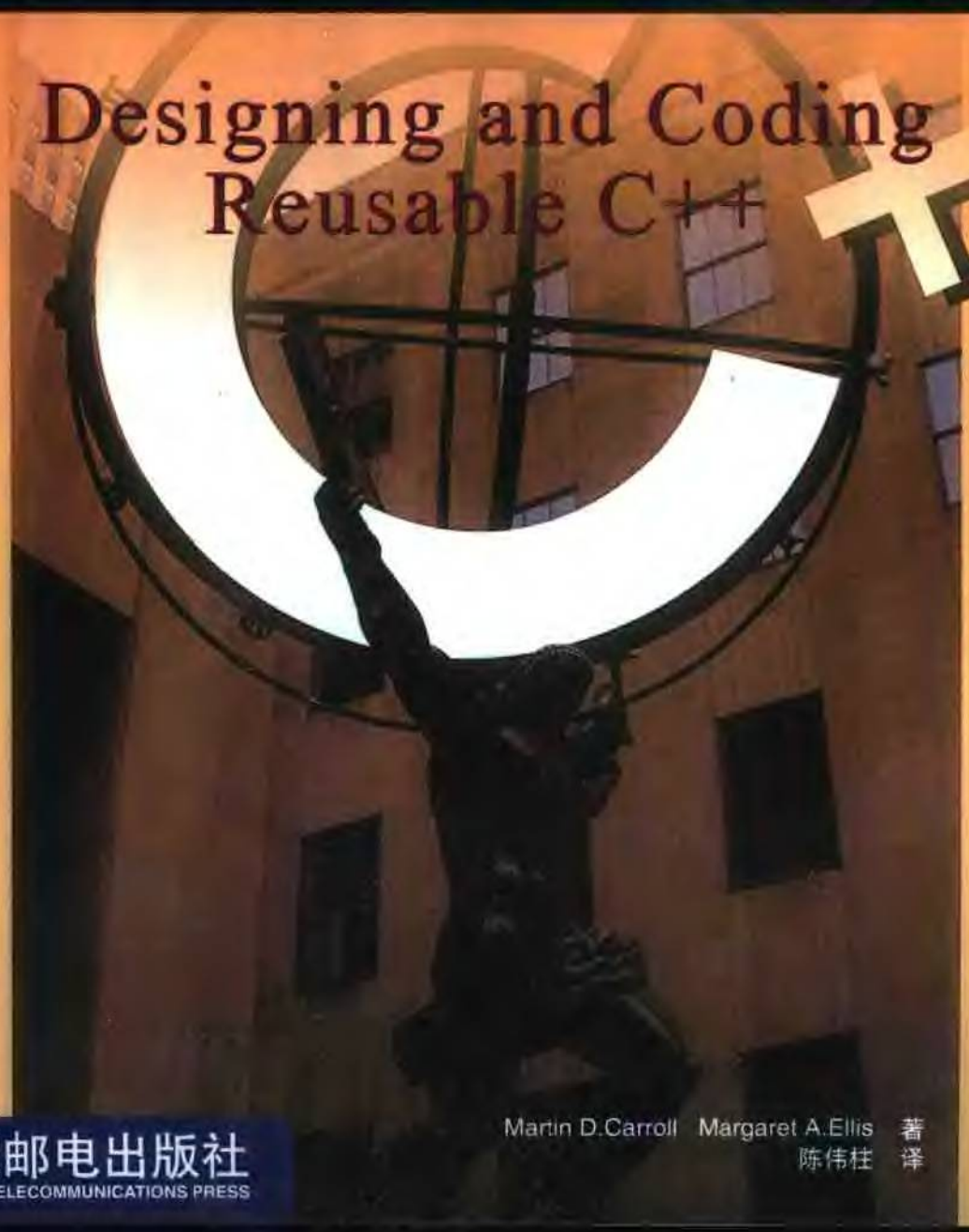
C 和 C++ 实务精选

Addison Wesley

C++代码设计与重用

Designing and Coding Reusable C++



Martin D.Carroll Margaret A.Ellis 著
陈伟柱 译

人民邮电出版社
POSTS & TELECOMMUNICATIONS PRESS

封面设计：胡平利

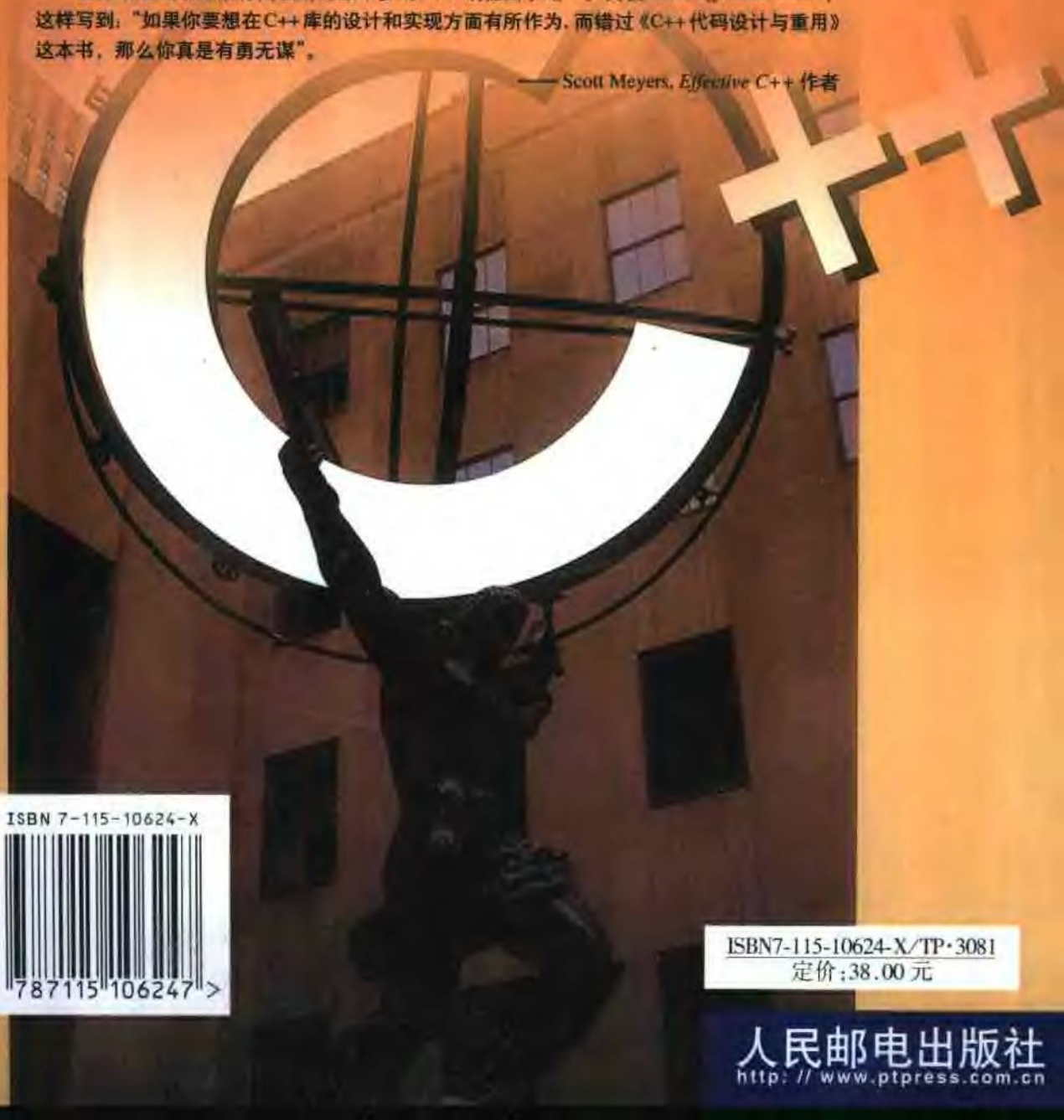
C和C++实务精选

C++代码设计与重用

学习C++库设计的宝贵经验
提高编写可重用代码的能力

这本书是我乐意推荐阅读的为数不多的C++编程图书之一。我在*More Effective C++*中这样写到："如果你要想在C++库的设计和实现方面有所作为，而错过《C++代码设计与重用》这本书，那么你真是有勇无谋"。

——Scott Meyers, *Effective C++*作者



ISBN 7-115-10624-X
9 787115 106247 >

ISBN7-115-10624-X/TP·3081
定价：38.00元

人民邮电出版社
http://www.ptpress.com.cn

C和C++实务精选

C++代码设计与重用

Martin D.Carroll
Margaret A.Ellis 著
陈伟柱 译

人民邮电出版社

C 和 C++实务精选

C++ 代码设计与重用

◆ 著　　Martin D.Carroll　Margaret A.Ellis
译　　陈伟柱
责任编辑　陈冀康

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
读者热线　010-67132705
北京汉魂图文设计有限公司制作
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：800×1000　1/16
印张：18.25
字数：400 千字　　2002 年 11 月第 1 版
印数：1-4 000 册　　2002 年 11 月北京第 1 次印刷

著作权合同登记　图字：01－2002－1550 号
ISBN 7-115-10624-X/TP・3081

定价：38.00 元

内容提要

本书全面展示如何使用C++编写可重用的代码，从而提高程序员的开发效率。

全书分为12章。包括重用性基本概念、类设计、扩展性、效率、错误、冲突、兼容性、继承、移植性、程序库等和重用相关的诸多话题。每一章的最后，通过总结和练习帮助你巩固概念、加深理解，参考文献和相关资料为你指明了深入学习的方向。

本书适合有一定C++经验的程序员阅读，也可供以提高代码重用性为专门学习方向的读者参考。

作者简介

Martin Carroll 是 AT&T 贝尔实验室的技术人员，他曾经用好几年的时间致力于设计和实现可重用的 C++程序库，包括 AT&T 标准组件库（Standard Components Library）。他在 Rutgers 大学获得计算机科学博士学位。

Margaret Ellis 是 *The Annotated C++ Reference Manual* 的合著者（另一个作者是大名鼎鼎的 C++之父 Bjarne Stroustrup），她主要致力于 AT&T 贝尔实验室、UNIX 系统实验室和美国 Novell 公司的编译器开发。她曾获得加州大学计算机专业的硕士学位。

中文版序

最早知道这本书，是在 Scott Meyers 著名的 *More Effective C++*书籍推荐列表里。在谈到这本书的时候，Meyers 说："有意撰写程序库的人，若没有读过此书，那只能是匹夫之勇。"而 Francis Glassborow 则说，大多数程序库设计者应该搬过小板凳来，像小学生那样学习这本书。

用 C++的人，历来以能撰写出重用性好的程序库而引以为傲。然而，撰写可用的程序库已属不易，撰写可重用的程序库更是专家级任务，以至于有人感叹道，重用是 C++程序员们的"圣杯"——总在嚷嚷着，却始终得不到。其实何止 C++如此，在其他语言中，重用又何尝不是令人胆怯的挑战！只不过 C++编译语言的本质与人们对 C++程序性能的严苛要求，使得用 C++编写可重用组件更为困难。这样一个艰难的主题，敢于涉足已经是非常有勇气了，而能得到 Meyers 等人如此高的评价，则非 C++中的顶尖人物倾力而为不可。事实上，作者 Martin Carroll 和 Margaret Ellis 正是这样的顶尖人物。他们是 AT&T 实验室的研究人员，从 20 世纪 80 年代中期开始就处于 C++演化和发展的中心，长期从事编译器和基础库的开发工作，积累了普通程序员无法比拟的经验。特别值得一提的是，Margaret Ellis 曾与 Bjarne Stroustrup 合作撰写了 C++早期的经典著作 ARM（*The Annotated C++ Reference Manual*），是 C++社群里最受尊敬的巾帼英雄之一。在这本书中，两位作者把自己多年来撰写可重用库的经验和教训高浓度地总结起来，给出了很多具有深刻洞察力的建议。虽然从此书英文版出版至今已有几年的时间，但是书中的真知灼见仍然值得我们一再学习体会。我甚至认为，即使你不打算撰写程序库，甚至不是 C++程序员，认真阅读这本书都是很有好处的。毕竟这本书来自具有丰富实战经验的顶尖专家之手，而这样的书是在是太少了。

译者陈伟柱是我的好朋友，在 C++、Java、模式和重用等方面都有比较深刻的理解。他对待此书翻译工作的态度极为认真，正是这一点令我对这本书的翻译质量很有信心。原书作者是以技术扎实而谦恭简和著称的，他们的文字也是平实简练而深富内涵的，我认为伟柱的为人和他的文字也具有这样的特点。因此，我衷心希望读者也能以平实谦和的心态认真、扎实地阅读和学习此书，并祝愿大家能有满意的收获。

孟岩

2002 年 10 月于北京

译者的话

这是一本令我受益匪浅的书，我相信很多读者也会有相同的感受。

重用——可以说是一个永恒的话题，只要有软件工程的继续存在，重用就必定会有一席之地，也会成为软件工程师或者程序员们关注的话题。重用的历史和重要性毋庸多言，几乎每本软件工程的书都会叙述重用的概念和大体内容；但是，现今关于重用的中文书，大多对重用的理论和优点介绍得非常详细，而很少涉及到如何设计与编写可重用代码的内容，给人一种隔靴搔痒、不着痛处的感觉。所以说，对于这本书，看完之后，我想说的一句话就是："它终于来了。

重用有3个基本的问题，一是必须有可以重用的对象，二是所重用的对象必须是有用的，三是重用者需要知道如何去使用被重用的对象。解决好这3个方面的问题才能实现真正成功的软件重用。本书的大体内容就是依据前面这两个问题，从类的层次体系设计、扩展性、效率、兼容性、移植性等方面详细叙述了可重用代码的设计原则，深入浅出地讨论了当重用性和上述的诸多特性发生冲突的时候，应该如何取舍，从而达到程序员或者程序库设计者预期的目的；而且，本书的第11章还针对上面的第3个问题给出了编写可重用代码文档的要求，为成功的重用代码设计加上了不可或缺的一环。书中的具体内容第1章已经详细给出，这里我就不再赘述。

本书的写作风格很有特色，它并不会教你要如何做，也不会给你叙述哪种设计与编码才是最佳的途径——金无足赤，永远为最佳的选择本来就是子虚乌有。作者凭借多年的编译器开发和可重用代码设计的经验，详细地阐述了在编写可重用代码的过程中一定会碰到的许多抉择，在阐述中还重点指出了许多会被普通开发者忽略的考虑因素；然后在不同的上下文中详尽地比较各种设计选择的优劣，让读者知其然更知其所以然，给致力于设计和编码的程序员点亮了一盏通向成功重用的明灯。

对于此书，我不敢说它弥补了国内重用书籍的空缺，但就程序库设计方面而言，它确实有着不可替代的作用，衷心希望每个致力于程序库设计的读者，都可以从此书获益。

翻译过程中，译者尽量以平实无华的语言来对待这本技术书籍；在力求技术准确的同时，更加注重上下文的连贯性，希望我的努力能给你带来一个顺畅的阅读过程。

译者

2002年10月

前言

一切事物都将得到检验并因此被称为问题。
——Edith Hamilton

这本书的主要目的在于：展示如何以C++编程语言编写可重用代码——就是说，根据不同的需要，在不经过修改，或者经过很少修改的前提下，可重用代码可以很容易地应用到5个、50个甚至500个程序当中，而且这些程序往往是不同程序员编写的，可能运行在不同的系统上。在整个阐述的过程中，我们的目的并不在于争论是否所有的代码都是可重用的，也不在于说明可重用代码能够解决所有的程序问题。显然，不论是对程序员而言，还是对可重用代码本身而言，提高代码的重用性都是需要代价的；通常只有当我们有理由相信所给代码在将来有可能会被重用时，我们才会付出这些重用的代价。因此，本书的目的在于详细分析重用性的这些代价，于是当你面对是否编写可重用代码的选择时，可以从容地做出明智的决定。

关于本书

本书主要面向的读者是：那些希望从书中包含的许多深层C++编程见解中受益的读者，或者是那些需要或希望学习如何编写可重用代码的读者。在论述过程中，我们假设读者已经知道如何编写正确的C++代码。

C++语言至今还没有经过标准化（译注：本书写作于1995年，C++于1997年标准化），任意两个不同的编译器实现支持的语言几乎都是不同（稍微不同或者相差很大）的。当我编写这本书的时候，并没有一个编译器实现可以完全支持ANSI/ISO C++标准中最终定义的整

个语言特性；而且，就算对同类型的编译器而言，前后版本实现的语言特性也不尽相同。于是，编写一本对所有编译器都适用的书是很困难的，或者是不可能的。因此，当我们讨论或使用一些不能被主流 C++编译器所实现的特性时，我们将会另加说明。

当声明本书中的代码例子被认为合法时，我们所指的合法性是以 1994 年 9 月份 ANSI/ISO C++的工作文件（有时候也称为“标准草案”）[ANS94]为依据的。而且，在我们的代码例子里，我们将尽量避免使用那些我们认为在最终 ANSI/ISO C++标准公布之前，很有可能会被删除或者进行重大修改的语言特性。

要跟踪 C++语言的演化，有几个资源是可以利用的。Internet 讨论组 comp.lang.c 和 comp.std.c++主要致力于 C++的讨论。互联网中还有一个关于 C++的主页：

```
http://info.desy.de/user/projects/C++.html
```

而由纽约 SIGS 出版社定期发行的 *C++ Report*，也经常会刊登许多 C++程序员感兴趣的文章。至于学习 C++的书籍，Lippman 的[Lip91]和 Stroustrup 的[Str91]是最佳的学习教材。

任何关于重用性的完整讨论都会涉及到软件开发的一些其他主题，如接口的设计、实现的效率、可移植性、冲突等等。在书中的论述中，我们假设读者对软件开发已经有所了解；因此，我们只注重于那些应用于可重用代码的主题，而不是那些应用于一般软件开发的主题。

“每本教材都会撒谎”，这是我们老师非常喜欢的一句谚语。在这里，她实际要表达的意思是：每本教材为了教学的简化性和明确性，都必须简化书中的内容，从而都会歪曲所要表达的信息。显然，即使在需要简化的前提下，我们还是应该用抓住众多细节间的本质信息的方法来表述每个主题，这本书也不例外。于是，对于某些主题，即使有些读者已经对这个主题的方方面面都有所了解，我们还是希望他们能够和我们一起共同探讨该主题的内容。而且，对于一些主题内容的省略，我们也希望能够得到这些读者的谅解。

选择与建议

设计和编写可重用代码将会涉及到许多选择（即在多种实现方案中选定某种方案），而某些选择的决定又是左右为难的。通常，对于这些选择，并没有完全确定或者正确的答案——无论你做出哪个选择，每个选择都是有代价的。我们将讨论各种不同选择方案的优点和缺点，从而让将来可重用代码的编写者可以从容地做出明智的决定。

有时，对于某个给定的决定，我们会认为某种风格或者方法要优于其他所有的风格或者方法。于是，当遇到这种情况时，我们会明确地建议采用这种风格或者方法，并给出支持这种风格或者方法的详细理由。然而，在大多数情况下，只有 C++程序库的设计者，才能决定哪种方法对程序库用户而言是最好的。

这本书的一个目的在于：详细并且清楚地给出编写可重用代码的大多数重要选择。如果我们忽略了某些重要的选择，那么我们愿闻其详。

书中大多数关于选择的讨论都会提到效率。虽然我们并不认为效率是所有可重用 C++代

码最重要的设计目标，因为对于许多程序库设计而言，譬如扩展性、灵活性等其他特性将会比效率更加重要；但我们在文章中又处处提到效率，这是因为效率将会和可重用C++代码的其他每个可取的特性都相互制约。既然对效率已经花费了这么多笔墨，我们也希望可以给出所有和效率相互制约的因素。

代码例子

我们希望这本书的价值在于：与刚开始读这本书之前相比，当你读完这本书的时候，你将知道更多关于如何编写可重用代码的知识。因此，我们使用了大量的例子，而且许多例子的代码是来自于现实中已经存在的代码，因为，那些不是取材于真实代码的例子，不足以阐明实际编程中会出现的问题，也不足以说明实际编程中使用的各种技术。在这里，我们只是为了阐明重用性的目的而利用这些代码。

为了节省篇幅，成员函数的主体通常都是在类的声明中给出；即使在重用性的代码设计中，这些函数也不会被实现为内联函数。例如，读者不应该从下面的类定义语句中得出：函数f是内联函数（就是说实际上f不是内联函数）的结论。

```
class Z {
    void f() {
        //...
    }
};
```

（关于如何决定某个函数是否应该实现为内联函数，我们将在4.4.1小节讨论这个问题）

当提到模板成员函数的时候，在不致引起混淆的前提下，我们通常都会省略模板参数。例如，我们将把下面代码中的成员函数f:

```
template<class T>
class X {
    void f();
};
```

表示为X::f，而不是X<T>::f。

通常，现实程序库例子中给出的类大多是模板，而在书中，为了能够使阅读更加容易，并且有利于更加清楚地表述每个主题，我们就进行了简化。相似地，现实程序库中的嵌套类也并不都如嵌套语法所定义的那样，但是，当我们在书中给出一个嵌套类的时候，我们将不使用前置声明语法。例如，我们将如下编写嵌套类：

```
class X {
    class Y {
        //...
    };
//...
};
```

而不是如下编写嵌套类：

模板函数：我们不使用这个术语。

例如，考虑下面的类模板：

```
template<class T>
class X {
    //...
};
```

类 X<int>和类 X<char>就是这个模板的两个特化。另外，特化通常是隐式定义的，如：

```
template<class T> class X { /*...*/ };
X<int> x;        //X<int>就是隐式定义的.
```

偶尔，某些C++程序员也会显式定义特化，例如：

```
template<class T> class X { /*...*/ };
class X<int> {/*...*/};      //X<int>在这里显式定义.
X<int> x;
```

而许多人把特化这个术语仅仅看成是显式特化，这是不正确的。虽然我们定义的这些术语都是非常相似的，但我们相信，ANSI/ISO 很快将会给出这些术语很好的定义。

通常，在不会引起混淆的前提下，我们会使用“类”来代替“类模板”，“函数”来代替“函数模板”。

致谢

在这本书的创作过程中，许多人通过多种方式支持过我们。我们在这里对他们表示衷心的感谢。

这本书的评审者所作出的贡献和该书的价值几乎是等同的。我们特别感谢那些对我们要求严格的评论者，他们总是相信我们可以做得更好。这其中包括 Tom Allocco、Manuel Bermudez、James Coggins、Keith Gorlen、Tony Hansen、Chris Hornick、Peter Juhl、Brian Kernighan、Andrew Koenig、Eason Kung、Rao Kuimala、Doug Lea、Stan lippman、Tom Lyon、Glen McCluskey、Barbara Moo、Rob Murray、Jishnu Mukerji、Scott Myers、Steve Pendergrast、Ed Schiebel、Jonathan Schilling、Jonathan Shopiro、Bjarne Stroustrup、Steve Vinoski、Jedy Ward 和 Clay Wilson。

我们也收到了许多读者很有用的反馈信息，他们是 Dag Bruck, Rich Kempinski, Josee Lajoie, Deborah McGuinness, David C.Oliver, Jeffrey Persch, Ellia Weixelbaum 和两个分别叫做 Lars 和 Steve 的网友。

这本书的许多观点首次来自于我们设计 C++标准组件库的实践，这个组件库最初是由贝尔实验室开发的。因此，我们要感谢我们的同事们关于这个项目的许多有深度的讨论。他们是 John Isner, Andrew Koenig, Dennis Mancl, Rob Murray, Jonathan Shopiro, Alex Stepanov, Terry Weitzen 和 Nancy Wilkinson。

如果没有 C++，那么这本书（也包括其他的许多书）肯定是不会存在的。因此我们非常

感谢 Bjarne Stroustrup 给我们带来了这样一个我们最喜欢的编程语言。另外也感谢 ANSI/ISO C++标准委员会的所有成员，正是因为他们孜孜不倦的工作（往往都是自愿而且无功利的），才能使整个 C++社区生机焕发。

感谢在我们这本书的编写过程中，我们这本书的审订者——Marha Currie, Barbara Moo 和 John Spicer——感谢他们的支持和鼓励。也感谢我们的雇主——贝尔实验室和 Unix 系统实验室（现在是 Novell Unix 系统小组）——感谢他们给了我们编写这本书的机会，而且还感谢他们为我们提供的时间、硬盘空间、打印纸等等。

我们非常感谢来自 Addison-Wesley 的大力支持。其中，充满活力的 Tom Stone 和生性乐观的 Debbie Lafferty 都是很好的合作伙伴。另外，Lyn Dupre 给了我们最好的编辑指导，书的设计者 Juliet Silveri 的工作总是一丝不苟，Pat Daly 是一个很专业的编辑，Roberta Clark 对我们的草稿进行了校对。我们还要感谢 John Wait，正是因为他一贯坚持让 Martin 编写一本关于 C++和重用的书，才会有这本书的诞生。然后，也很感谢 Jim DeWolf，他为我们提供了暖气和空调，这笔支付也是不菲的，从而给我们带来了一个很好的环境和开端。

衷心感谢 Tom Reinhardt，他可以说是世界上最好的外科临床医学家。也感谢我们的朋友 Paul Lustgarten，他是我们以前的房东，很慷慨地允许我们时时占用他的电话线。还感谢 David Wooley，正是他让 Martin 在大学中进入到 BASIC 编程领域中来的。真诚地感谢 Marybeth 为我们拍了书背面的那张照片。

最后，但也是很重要的，我们真诚感谢我们的家人和朋友，他们和我们度过了这本书编写过程中的许多酸甜苦辣的日子，并且时时刻刻给予我们支持与关怀。

图书在版编目（CIP）数据

C++代码设计与重用/（美）卡罗尔（Carroll,M.A.），（美）埃利斯（Ellis,M.A.）著；陈伟杜译. —北京：人民邮电出版社，2002.11
ISBN 7-115-10624-X

Ⅰ. C… Ⅱ. ①卡…②埃…③陈… Ⅲ. C 语言—程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2002）第 076775 号

版 权 声 明

目录

第1章 重用性介绍

程序库是聪明的程序员最好的咨询室。
——George Dawson

在介绍重用性的这一章里，我们先描述什么是重用性，其中特别着重讨论代码提取技术。因为代码（仅供参考）提取是重用的原始形式，也正是它的不足促使我们对可重用代码下定义。接着我们反驳一些有关重用性的神话（对重用性错误的理解），并列举了阻碍重用性发展的障碍——技术性的和非技术性的。最后，我们对有志于编写可重用代码的程序员提出若干期望。

1.1 什么是重用性

许多相同操作都会在多个计算机程序里重复实现，例如：

- 对数组元素进行排序；
- 解答线性方程组；
- 实现一个从X类型到Y类型的映射；
- 解析C++代码；
- 从数据库检索数据；
- 和其他程序进行通信。

与其在每个程序里都设计和实现上面每个操作的相同代码，我们更愿意采用的方法是：只设计和实现这些操作的代码一次，然后再把这些代码重用于不同程序里。显然，已有的可重用代码，使每个应用程序不必从头写起，因为它（可重用代码）大大加速了应用程序的开

发，并且减少了编写和维护应用程序的费用。

代码重用已经不是一个新概念了。第一代高级编程语言 FORTRAN 的最初版本就有了一个可扩展的程序库，它可以实现输入、输出和格式化等操作。当 FORTRAN 语言新推出的时候，重用就已经不是什么时髦用语了，何况 FORTRAN 程序库就已经包含了许多可重用代码。另一个典型的例子是 UNIX 操作系统，它提供了许多相当有用的小指令，这些小指令可组合成用途广泛的新指令，从而通过这种方法来鼓励重用。

1.1.1 提取代码来作为重用

代码重用的最初形式就是从其他程序里提取代码。假设我们正在设计某个程序设计语言的编译器。在编译器实现的某个地方需要一个高效的哈希表来管理符号表，而我们又不想从头来设计和实现哈希表；这时，如果我们能够在其他的编译器找到适合的哈希表实现代码，我们就可以从这些编译器中提取代码，然后再拷贝到我们的程序中去。

提取在实际中应用得很普遍。但令人遗憾的是提取有着很多显著的缺点。第一，找到需要的代码相当困难。每个人所能提取的代码只能是所有存在代码的一小部分，因为提取代码实体要求程序员对代码具有存取权限和深刻的理解。所以程序员一般只是提取自己编写的那部分代码，或者就是同一个开发小组内其他程序员写过的代码。第二，把从其他程序里提取的代码用到自己的程序上，正确与否的把握性不大。第三，每个有经验的程序员都知道，由于要提取的代码和程序里其余的代码之间存在相互依赖性，让一段重要的、要提取的代码从包含它的程序里分离出来会是非常困难的。第四，提取的代码往往需要经过重大改写才能用于新的程序里。例如，提取代码里的名字可能会和新程序环境里原有的名字发生冲突；被提取的代码可能还认为它处于原先的执行环境里，实际上这样的环境在新程序里已经不存在了；被提取的代码在原先程序里可能由于使用方式的原因已经优化过了，但在新程序里它却没经过优化等等。

从头开始编写代码往往比提取代码显得更加容易——即使我们已经知道在别的程序里有我们需要的代码并且可以得到这段代码。

1.1.2 可重用代码的基本特性

提取的上述缺点是可以避免的，只要我们要提取的代码具有以下特性：

- 代码容易找到且容易理解；
- 对代码的正确性很有把握；
- 不需要从包含它的外层代码分开；
- 当把代码用于新的程序里时，不需要对代码进行修改。

这些特性是任何可重用代码都必须具备的最少特性。对特殊的可重用代码段，往往还必须具有其他的一系列特性，如高效性、可扩展性、可移植性等等。

程序库是可重用代码的集合。在 C++中，一个程序库是由（零个或多个）公共头文件、

（零个或多个）源文件、（零个或多个）目标文件和（零个或多个）用于实例化模板必需的文件的集合组合而成的。其中，目标文件经常是由一个或多个文件打包而成。这些被打包的文件，在一些系统中，我们称之为档案文件，在其他系统中我们称之为链接库（如动态链接库 DLL 就是一种特殊的链接库）。尽管链接库概念使用得比较普遍，但为了避免它和程序库发生混淆，我们在这里还是使用档案文件这个术语。为了使用一个程序库，我们就应该包含（#include）它的头文件，并使用系统提供的特定机制，来编译它的源文件和链接由编译产生的目标文件和档案文件。

1.2　重用的神话

关于代码重用出现了许多神话（荒诞的说法），这一节我们来反驳几个比较普遍的说法。

神话 1：重用可以解决软件危机

软件危机是指程序设计团体现今没有能力做到以下几点：编写解决复杂问题的程序，快速生成解决复杂问题的程序，正确编写这些程序并使这些程序的维护相当容易。

软件开发进步的迹象是显而易见的。一个很显然的迹象就是随着时间的推移，所谓的复杂问题的范围发生了改变。在 20 世纪 60 年代，编写一个 FORTRAN-66 编译器就被认为是一个非常复杂的问题；建构一个可以正确运行的 FORTRAN-66 编译器更被认为是非常费时的工作，并且最终的编译器产品的可维护性也远比不上今天的编译器。到了今天，编写一个正确并且可维护的 FORTRAN-66 编译器的过程已经是比较容易的了。但即使这样，还是没有人会把能够很容易地编写一个 FORTRAN-66 编译器视作软件危机解决的证据，尽管，编写 FORTRAN-66 编译器已经不再被认为是复杂问题了。

因为人们总会不断遇到更加复杂的问题，就算这些问题可以用相应的软件来解决，他们还是会认为是软件危机导致了这些复杂问题，这种观念不太可能很快改变的。然而，这并不意味着重用就一无是处了。实际上，当程序员重用的代码越多，他们的编程任务的复杂度就会降低得越多，原先复杂的问题就会变得越简单。

神话 2：所有的代码都应该是可重用的

我们从整本书可以看出：重用是有代价的。如果为了永远不会重用的代码而付出重用的代价，那将是毫无意义的。但请注意，我们一般只是在代码写好了之后才会发现这段代码实际上是可以重用的；这时，为了能够提高代码的重用性而改写这段代码，将比从一开始就写出可重用的代码困难得多。

神话 3：重用代码总是优于从头开始写代码

有时，从头开始编写正确的、可维护代码将比找到一段可重用的代码更加容易，特别是

小程序更是如此。例如，我们要实现一个把华氏温度转化为摄氏温度的C++函数，那么，从头开始编写这样一个函数就会比搜寻这个函数的可重用代码容易很多，即使这样的可重用代码是存在的。

神话4：面向对象的语言使得编写可重用代码更加容易

大多数用面向对象的语言使代码在大范围内被重用的程序员都只是认为，与非面向对象的语言相比，面向对象的语言使代码的重用更加容易。但却几乎没有人说过是面向对象的语言使重用更加简单。实际上，成功的代码重用主要是依靠技术、经验和孜孜不倦的努力，而不仅仅是语言。

1.3 重用的障碍

事实上重用是很难实现的——存在着非技术上和技术上的障碍。非技术上的障碍是指诸如组织结构、社会结构、程序设计文化等阻碍重用的事物。技术障碍是指程序设计本身阻碍重用的各种因素。

1.3.1 非技术障碍

为了理解重用的非技术障碍，我们来考虑重用对C++程序员自身的影响。例如，程序员在一个函数库里设计一个名为Widget的类。

类Widget的设计者应该怀疑设计可重用的Widget类是否会有用处

因为使代码能够被重用是需要时间和精力的，所以大多数程序员如果觉得他们的努力是没有价值的，就不愿意编写可重用代码。

类Widget的设计者由于编写和提供可重用代码应该有望得到回报和奖赏

因为花费时间和精力的事情如果不能得到回报，一般是不会有人愿意做的。但遗憾的是，编写可重用代码，程序员有时非但不能得到回报，而且还受到（一般是无意的）责难。

假定有两个程序员——Pat和Chris——在某家大公司的某个部门工作。Pat总是能够按时完成她的任务，但她编写的代码只有她个人或她所处的项目小组成员可以重用。而Chris花更多的时间编写可供整个公司所有程序员重用的代码。整个公司的程序员由于使用了Chris的代码将大大节约开发时间，但Chris往往因此而不能按时完成任务。于是，如果Pat和Chris的部门只注重于个人是否可以按时完成任务的话，那么Chris得到的将不会是奖赏，而是处罚。

必须有人来维护类Widget

一段可重用代码越是成功，越需要花更多的时间来维护；维护包括修改代码的错误，改

写代码使它符合新的要求等。如果 Chris 的代码能够在整个公司使用，那么她将花费大量的时间来回答有关代码的问题，分发代码和不断发展这段可重用代码。维护一段成功的可重用代码的所有时间可能要比编写这段代码的时间还要多；所以，如果维护可重用代码毫无回报的话，那么重用也只能昙花一现。

类 Widget（假设已经设计成可重用的）的最终用户应该觉察到 Widget 的存在性和可获得性

一个大程序需要很多可重用的代码段，但在整个世界里搜寻符合要求的可重用代码将会花费很多时间。所以程序员有时应该质疑搜寻一段特殊的可重用代码的成功率。

很明显，一个组织应该公布它的可重用代码(或者向内公布，或者向外公布，这依赖于它希望有多少用户使用这些代码段)。程序库必须提供足够的文档说明，并且提供一些可以很容易找到这些程序库和程序库里可重用代码段的工具。

类 Widget 的最终用户必须能够得到类 Widget

得到类 Widget 可能只需简单地拷贝几个文件，或许可以从一张包含有类 Widget 源代码的光盘里获取源代码，然后再根据这些源代码中创建类 Widget。遗憾的是，分发文件和创建类在实际上经常不能顺利进行，这也就成了重用的一大障碍。

类 Widget 的重用必须没有法律障碍

一些软件可以免费提供给任何使用它的人，另外一些软件只有当你付了费，同意和签署了许可协议和保密协议之后才能使用。这种法律的约束也阻碍了重用的发展。

类 Widget 的最终用户应该由于重用了类 Widget 而得到奖赏

一些机构（幸运的是现在已经很少了）是以程序员每天编写代码的行数来衡量程序员能力高低的。若以自己编写的代码的数量来说，重用代码的程序员是比不上没有重用代码的程序员的。因此，这种赏罚制度也扼杀了重用的发展。

当今对于重用的宣传已经引起了程序设计社区对重用潜在优点的关注，尽管有些宣传抱有过多的期望，但我们仍然希望，实现重用性潜在优点的期望会使一些组织改变对待重用的态度。他们应该是鼓励重用，而不是限制和制约重用。

1.3.2　技术障碍

重用性具有许多大大小小的技术障碍，主要的障碍如下：

可重用代码会在很多环境（context）下使用。其中，程序代码的环境是指：

- 使用这段代码的某个程序；
- 程序执行时候，代码是如何被调用的；

- 创建代码的平台（包括机器类型、系统接口和可用资源等等）；
- 外层代码执行的平台（外层代码指本身包含这段可重用代码的那些代码）。

显然，一段代码能够在越多的环境下使用，它的重用性就越好。

我们不可能对环境信息了如指掌

可重用代码被调用时，我们对调用环境各个方面的信息都是陌生的。我们一般不能检查调用这段代码的用户程序，我们因此对程序的执行情况往往一无所知。或许我们只能在程序调用我们的代码时，估计一下代码调用需要的系统资源的情况。

用户需求经常是互相冲突的

即使我们可以通过某种方法了解到可重用代码被调用时的所有环境信息，但由于用户需求常常互相冲突，我们仍然会面临着重大的设计抉择。在这时，程序库设计者通常不得不做出某种抉择，而这种抉择往往是不能令所有的用户都感到满意的。

我们提供的功能不可能面面俱到

如果我们的用户有互相冲突的需求，我们为什么不能在可重用代码里提供所有的功能呢？答案是很简单的，因为由此引起的庞大的代码实体将会导致昂贵的开发费用，而且很难使用。

环境总是不断变化的

更加糟糕的是，使用可重用代码的环境会随着时间的推移而不断变化。新用户要把这段代码用于新的程序里，老用户为适应新的需求要更改原先的程序。因此，可重用代码的设计者必须尽力预测将来可能的变化，从而编写出可以适应将来变化需求的可重用代码。但是，对将来的预测往往是非常困难的。

1.4 希望是否尚存

读了这么多关于重用的障碍之后，你可能会怀疑重用性是否还有存在的希望？毫无疑问，希望是存在的。首先，可重用程序库（既有商用的，也有个人使用的）的大量存在和普及就说明了软件重用是完全有可能的。现今就有几个高质量多用途的程序库存在。例如实现链表、集合和字符串等数据结构的程序库；支持开发用户图形接口和窗口应用程序的程序库；另外还有很多专业程序库，例如数据库程序库、远程通信程序库、股票行情和金融分析程序库，物理数字处理程序库，实际上还有很多。相对于没有存取权限的程序员而言，对这些程序库拥有存取权限的程序员将可以更容易地开发出高质量的软件。

其次，没有程序库可以做到面面俱到。充分考虑你面向的应用领域、应用程序的领域和用户的需求。如果你的应用领域只是诸如 10 个执行数字处理这样的应用程序，你就没有必要

确保你的代码在每个已经写好的 C++程序里都可以重用。同样地，如果用户只注重程序的效率，那么我们花很多时间来提供可扩展性和可移植性也是不可取的（可扩展性和移植性一般都会影响效率）。再次，如果你多花些时间来做出正确的设计决定，那么你产生可重用代码的机会将会大大提高。下面是几个在设计基础程序库时需要注意的问题：

- 程序库的效率如何？
- 程序库在什么情况下才是可扩展的？
- 程序库的各个部分之间是如何进行耦合和解耦合的？
- 程序库的移植性如何？
- 程序库是否需要使用其他的程序库？

书中还讨论了程序库设计要注意的许多其他问题。

1.5　这本书能给我们带来什么

编写可重用代码可以使复杂的问题变得比较简单，但编码过程是非常困难的。这本书不会也不能让这困难的过程变得格外简单，这本书也没有提供能让每个 C++程序员都可以很轻松地编写出可重用代码的锦囊妙计。

针对每个希望编写出可重用代码的 C++程序员，这本书的每一章都讨论了一个或者多个他们必须理解的问题。理解了这些问题虽然不能使编写可重用代码变得相当简单，但可以让编写出可重用代码成为一种可能。

这本书的其余部分的结构如下：

- 当今流行的 C++程序库主要包含的是类的集合，因此我们在第 2 章开始讨论如何对类进行优化设计。
- 当今流行的 C++程序库在某种方式下是可扩展的，因此我们在第 3 章讨论扩展性。
- 在第 4 章，我们列举了一些编写高效的可重用代码的技术。
- 在第 5 章，我们讨论了有关错误的一些问题——在可重用代码里如何检测和避免错误，当错误产生时应该采取什么措施等。
- 在第 6 章，我们解释了如何解决在同一个程序里的可重用代码和其余代码之间的冲突问题。
- 第 7 章介绍了现在比较流行的兼容性问题。
- 第 8 章讨论了各种继承层次体系设计的优缺点。
- 第 9 章讨论了如何提高可重用代码的移植性。
- 在第 10 章，我们讨论可重用程序库是否可以重用其他程序库的代码。
- 在第 11 章我们解释如何给 C++程序库编写文档说明。
- 最后在第 12 章，我们讨论了其他的一些问题，如静态初始化、类耦合等等。

1.6 练习

在整本书里，困难的题目将会在题号后面标示（*），特别困难的题目会用（**）标示。

1.1 假设你在实现一个函数，这个函数在一个给定的数组中查找某个给定的值。

a. 在什么情况使用函数时，线性查找的实现比二分查找的实现具有更高的效率。

b. 为了能在任何环境下，你的查找算法都可以更加高效地执行，你应该如何（使用什么算法）实现这个函数呢？

c. 假设在这里，我们并不是查找用户给定的值，而是查找第一个0出现的位置，那么需要在什么样的条件下，线性查找才能比二叉查找更有效呢？

d. 假设我们现在的查找范围不再是用户给出的数组，而是一个具体的数组，但查找值仍然是用户指定的。假设函数is_prime为了判断某个小于1000的数字是否是素数，需要查找一组所有小于1000的素数。那么，以这个假设为前提，在什么样的条件下，线性查找才能比二叉查找具有更高的效率？

1.2 在这个练习和下面的练习里，我们将分析一个问题，它是由程序库设计者未能在开初就显式声明程序库的用户类型而引起的。

假设我们是可重用类Path的设计者，其中Path表示我们用户文件系统中的路径名称。例如，在UNIX系统中，典型的路径有：dir、dir/dir2/tmp和../tmp等；在Windows系统中，相应的路径应该写成：dir、dir\dir2\tmp和..\tmp；而在VMS系统中，这些相应的路径为：dir、[dir.dir2]tmp和[-]tmp。

假设当我们开始设计Path时，我们忘记了指定这个类的用户类型；那么，在设计过程中途，如果我们是UNIX程序员，我们可能就只会以UNIX路径的模式来设计Path：

```
class Path {
public:
    Path();
    Path(const String& s);
    //...
private:
    void canonicalize();
};
```

上面的代码中，缺省构造函数创建了一个空路径，另一个构造函数创建了一个规范形式的路径s。在规范形式里，.标志符已经被删除，..标志符也尽可能被去掉，多个连续的/被精减为单个/，剩下的/都删除掉了。于是，例如下面字符串

```
../../dir/./dir2//other_dir/..tem.c/
```

的规范形式是：

```
../../dir/dir2/tem.c
```

a. （*）如果用户让Path的构造函数自动规范化所给的字符串，那么对于什么样的UNIX

系统用户，才会出现不可取的行为？（提示：何时 dir/../some_file/和 some_file 表示不同的文件？）

b. 对于 Windows 用户而言，为什么第一个构造函数的行为是错误的呢？

1.3　为了能够同时适应 Windows 系统程序员和 UNIX 系统程序员，我们可以改变 Path 类的设计，让用户指定作为路径分隔符的字符：

```
class Path {
public:
    Path(const String& s, char separator);
    //...
};
```

于是，UNIX 系统上的用户必须指定‘/’为分隔符，而 Windows 用户必须指定‘\\’（这实际上是一个转义后的反斜杠符）分隔符。那么这个改变会给现在的 UNIX 用户带来什么样的问题呢？你是否可以在避免这种问题发生的前提下，改变 Path 类的设计呢？

1.4　假设我们现在希望为 VMS 用户提供 Path 类。在 VMS 系统中，没有特殊的字符被认为是路径分隔符；于是，练习 1.3 设计的 Path 接口并不适用于 VMS 的用户。为了改善 Path 的设计，我们必须再次改变 Path 的实现：

```
class Path {
public:
    enum Style { UNIX,Windows,VMS};
    Path(const String& s, Style style = UNIX);
    //...
};
```

那么对于当前的 UNIX 用户和 Windows 用户，这个改变会导致什么样的问题呢？

1.5　给出设计下面 4 个类的方法：Paht、Unix_path、Windows_path 和 Vms_path，后面 3 个类派生自第 1 个类，并且用户可以如下编写代码：

```
Vms_path z("[dir]tmp");
Unix_path p("/dir/tmp");
Windows_path q("\\dir\\tmp");
```

假设不存在早期版本的 Path 类的用户，那么与练习 1.4 的设计相比，请给出这个设计的一个缺点和一个优点。

1.7　参考文献和相关资料

我们还没有看到软件重用性与编程语言无关的介绍，可能是因为，如果不借助于某种特殊的编程语言，很难编写许多和重用性相关的有价值的（或者有意义的）内容。Cline 和 Lomow[CL95]、Meyers[Mey92b]和 Murray[Mur93]的第 9 章都讨论了某些和这本书相关的内容。

Fontana 和 Neath[FN91]认为：某些程序员即使在代码可以重用的情况下，还是倾向于用

提取技术。

Tracz[Tra88]和 Plauger[Pla93]都讨论了某些重用的神话。

讨论 C++程序库设计的书籍有：Booth 与 Vilot[BV93]、Coggins[Cog90]、Lea[Lea93]、Keffer[Kef93]、Koenig[Koe91]和 Stroustrup[Str93]。Musser 与 Stepanov[MS94]提供了一个设计得很好的 C++程序库实例。Staringer[Sta94]给出了另一个成功的可重用实例。

IBM 系统杂志[IBM]的第 32 卷第 4 期的主题就是重用的非技术障碍（和其他某些可重用主题）。Fafchamps[Faf94]讨论了几种不同的、有利于和不利于重用的组织结构。Lim[Lim94]记录了一个组织可以影响重用性的最大程度。

讨论重用性的合法问题已经超出了本书的范围，Will、Baldo 与 Fife[WBF91]对这些问题有详细的描述。

第2章 类的设计

我并不相信阶级的差异，但幸运的是，我的管家并不这样认为。
——Marc 写于伦敦《时代》杂志的漫画

大多数 C++程序库主要是由类（和模板）的集合组成。每一个可重用的程序库，它的类必须是经过精心设计的。在这一章里，我们论述几个对可重用类的设计很重要的主题：抽象化、正规函数、nice 类、类接口的一致性，类转型和类接口中 const 关键字的使用。

有人认为，存在着一个最小的标准接口，所有的类都应该符合（实现）这个标准接口。我们驳斥了这种观点；尤其是，我们反对提供浅拷贝操作和深拷贝操作。

2.1 抽象性

每一个 C++类——不论可重用与否——都应该表示某种抽象。例如类 Rational 可以表示有理数的集合，类 Car 可以表述车的概念，类 Parser 可以表述 C++解析器的概念。当设计一个类的时候，我们要做的第一件事就是定义类所要表述的抽象。

一旦定义了类要表述的抽象，我们就可以有代表性地用某种方法来实现这个抽象。例如，我们可以这样来实现 Rational 类，定义两个整数，并且要求第二个整数不能为零。

```
class Rational {
private:
        int num;
        int denom;
public:
        //...
};
```

或者，我们可以通过十进制反复展开的方法来表述一个有理数[1]。

将类的抽象从它的实现分开是很有裨益的。理由有二。第一，简化了类的抽象。当对类进行抽象的时候，我们会忽略那些对当前设计目的不重要的细节。譬如对 Car 驾驶者而言，诸如车的划痕数量、凹痕数量和是否有小的后备仓等细节就显得无关紧要了。因此对 Car 抽象的完整描述就不应该包含这些细节，Car 驾驶者也会把这些细节抛于脑后。另外，类的文档化抽象是类的设计者和使用者之间契约的一部分。可以说，类的抽象越简单，所有团体对这个契约的解释越趋向一致。

第二，把抽象从实现分开，将使抽象的实现更富有灵活性。例如，某个程序库可能提供了上面所述的类 Rational 的两种实现方法。但是，如果我们把抽象从实现分开，并且程序库不提供上面任何一种实现方法，只是提供类 Rational 的设计方法；那么，用户就可以很容易地提供他们自己的实现方法。

恰如类应该表述抽象一样，函数也应该表述某种抽象行为。这就是说，几乎每个函数的语义都应该只根据操作对象的抽象值来定义。考虑下面的函数：

```
Rational operator*(const Rational& r, const Rational& s);
```

如果这个函数要实现乘法运算，我们可以这样来定义它的语义：

Return the product of r and s. (返回 r 和 s 的乘积)。

请注意这个语义的定义，它并没有提及返回的 Rational 中的 num 和 denom 是否会以简化形式存在（就是说，当 num 和 denom 的值本应分别为 3 和 1 的时候，它们是否会相应地以 9 和 3 这种非简化形式存在；对这点，这个语义并没有说明）。实际上，关于 Rational 是由 num 和 denom 来表述的这个细节，并不属于抽象的一部分。

2.2 正规函数

对所有提供它们的类而言，某些函数应该具有相同的语义。考虑类 Rational 的拷贝构造函数：

```
class Rational {
public:
    Rational(const Rational& r);
    //...
};
```

上面的操作将会构造一个 Rational 对象，它的值等同于对象 r 的值（我们所说的值总是指抽象值）。我们认为，每个类的拷贝构造函数都应该具有这样的语义，就是构造一个和它的参数等值的对象。尽管 C++没有——也不能——强制拷贝构造函数遵循这种约束，但每个设计良好的 C++类都应该遵守这种约束。

如果在所有设计良好的类中，某个函数的语义都是相同的话，我们就称这个函数为正规函数。C++中的正规函数有：

[1] 即把一个有理数展开成千位、百位、十位、个位和小数位等，然后用各个位上的数字来表述这个有理数。

- 拷贝构造函数；
- 析构函数；
- 基本的赋值运算符（指类 T 的赋值运算符，它的唯一参数的类型是 const T& 或 T）；
- 相等运算符和不等运算符。

下面的类 T 显式声明了（上面）这些正规函数：

```
class T {
public:
    T(const T& t);
    ~T();
    const T& operator=(const T& t);
    //...
};
bool operator==(const T& t1, const T& t2);
bool operator!=(const T& t1, const T& t2);
```

在这里，我们将相等运算符和不等运算符声明成全局函数。当然，它们也可以作为类 T 的成员函数。（bool 类型是 C++新增的类型，它有真和假两种值，真值用新关键字 true 来表示，假值用新关键字 false 来表示。）

正规函数的语义如下：

- `T::T(const T& t);`

创建一个 T 对象，它的（抽象）值和 t 对象的（抽象）值相等。

- `T::~T();`

销毁 T 对象。

- `const T& T::operator=(const T& t)`

用参数 t 的值给这个新对象赋值，并返回一个此对象的引用。

- `bool operator==(const T& t1, const T& t2);`

当且仅当 t1 和 t2 具有相同的值时，返回 true。

- `bool operator!=(const T& t1, const T& t2);`

当且仅当 t1 和 t2 具有不同的值时，返回 true。

上面这些正规语义都是抽象的；当然，它们也就有许多种合理的底层实现方法。（实际上，对于 operator==和 operator!=这两个运算符，为了确保能够准确无误的实现，它们其中的一个是根据（通过调用）另一个的实现细节来实现的。）即使上面所给函数是由编译器隐式生成的（构造函数，析构函数和赋值运算符是可以由编译器隐式生成的）[1]，或者是成员函数，而不是全局函数（双目运算符往往实现成全局函数），这些语义还是成立的。

对所有设计良好的代码，它们所有正规函数的语义都是相同的，所以，（严格说来）给函

[1] 关于编译器如何实现这 3 个函数，具体请参阅 *Inside the C++ Object Model*，但有一规则就是：拷贝构造函数、析构函数和赋值运算符这 3 个函数，要么都由编译器隐式生成，要么都由用户提供具体实现，具体参阅 Herb Sutter 在 *CUJ* 的 Conversation.

数的语义添加帮助文档是没有必要的。然而，一些过于谨慎的C++程序库设计者为了确保用户可以了解添加的正规函数，往往在很多地方添加了正规函数语义的帮助文档，但这样做的意义并不大。

2.3 Nice 类

我们都知道类会提供某些函数，这些函数要么是在类的代码中被显式声明为公共的（public）或保护的(protected),要么是由编译器在程序需要这些代码时隐式生成的。例如，下面这个类：

```
class X{
public:
    X();
    void f();
};
```

它提供了一个缺省构造函数、函数 f、一个拷贝构造函数、一个赋值运算符和一个析构函数。而且最后 3 个函数会在程序需要它们的时候由编译器自动生成。

请考虑下面这个通常有用的函数：

```
template<class T>
void swap(T& t1, T& t2){
    T t=t1;
    t1=t2;
    t2=t;
    }
```

如果 x1 和 x2 是某个类 X 的两个对象，那么我们就可以通过调用 swap 函数来交换它们的值：

```
swap(x1, x2);
```

为了使上面这行代码能够通过编译，类 X 必须提供一个拷贝构造函数、一个赋值运算符和一个析构函数。

现在让我们再来研究另一个通常有用的类：

```
template<class T>
class Array {
private:
    T* rep;
public:
    Array(int size) {
        rep = new T[size];
        //...
    }
    int size() const;
    T operator[](int i) const;
```

```
    //...
};
```

基于上面代码段，如果想创建一个含有 20 个 X 对象的 Array 数组，我们可以这样编写代码：

```
Array<X> array(20);
```

为了使上面这行代码可以顺利通过编译，X 必须提供一个缺省构造函数——否则 Array 构造函数中的 new 调用将会是非法调用。

最后，考虑下面一个通常有用的函数：

```
template<class T>
bool linear_search(const Array<T>& a, const T& t) {
    for(i = 0; i < a.size(); ++i)
        if(a[i] == t)
            return true;
    return false;
}
```

为了在 array 数组中查找一个值为 x 的 X 对象，我们可以编写如下代码：

```
if(linear_search(array, x))
    //...
```

为了使这行代码能顺利通过编译，类 X 必须提供相等运算符，或者能够转型一个新的类型，而该类型提供了相等运算符。

下列函数的使用范围非常广；以至于，如果一个类提供了下列所有函数，我们就称这个类为 nice 类：

- 缺省构造函数；
- 拷贝构造函数；
- 赋值运算符；
- 相等运算符；
- 析构函数。

非 nice 类限制了类的使用性，这种限制有时候是很严重的。因此，可重用类应该尽可能地设计成 nice 类。

2.4　存在最小标准接口吗

一些专家（如[RC90]里的 Riel 和 Carter）在对类进行深入研究之后，主张所有的类都应该提供某个最小标准接口。但究竟应该提供哪种最小标准接口呢？不同专家的建议往往又大相径庭。所有建议的标准接口除了包含 nice 函数（指 nice 类提供的函数）外，还包含诸如输入输出函数、哈希函数、以字符串返回类名的函数、浅拷贝和深拷贝操作等。

即使提供某个最小标准接口的动机是好的，但如果试图对所有的类都定义这个最小标准接口，那就是很不可取的。没有任何一个函数是所有类都必须提供的，下面的论据就说明了

这个观点：对任何建议的某个最小标准接口函数，肯定会描述出某个类，它根本不需要提供这个函数。

如果某个类是最小标准接口函数的反例，通常是由于以下3种原因中的一种：未能给出函数切合实际的语义，函数可能没有实现的价值，或者，就算函数刻意实现，但它带来的坏处往往多于好处。

在这里，由于篇幅的限制，我们不可能给出每个被建议为最小标准接口函数的反例，但我们将给出析构函数以外的所有nice函数的反例。由上可知，如果可以给出这些函数的反例，我们就有充分的理由相信：除了析构函数之外，其他所有函数都有不能作为最小标准接口的函数。

2.4.1 缺省构造函数

考虑下面具有特殊用途的内存配置器(allocator):

```
class Pool {
public:
    Pool(size_t n);
    void* alloc();
    void free(void* p);
    //...
};
```

如上所示，Pool(n)构造函数高效地分配和回收n个字节单位的内存块；Pool::alloc函数用来返回一个指针，这个指针指向一块至少含有n个字节的连续空白内存块。对同一个Pool对象q，如果p是由q通过调用alloc函数返回的指针，那么调用q.free(p)将会把p指向的内存块释放并返回给对象q[1]。于是，Pool::free函数的合理实现将是有前提条件的，这就是，它的参数的值（也就是指针指向的对象）必须是同一个Pool对象早先通过调用alloc函数的返回值。

但如果我们给Pool提供一个缺省构造函数，情况又会是什么样呢？

```
class Pool{
public:
    Pool();
    //...
};
```

由上可知，Pool的缺省构造函数必须创建一个可以高效分配和回收n个单位内存的对象，而不管这个n的值是多少。因为在缺省构造函数中，用户并没有给出n的值，所以如果我们（Pool类的设计者）必须决定应该对它分配多少内存块，又该使用哪一个具体的n值？

随机地选择一个数值来作为n值，譬如37，显然是错误的。那么，我们或许会考虑选择0或者1。然而，用户不太可能要求分配0或1个字节的内存。实际上，创建一个分配0或1个单位内存的Pool对象很可能会导致逻辑错误。这样，为类Pool提供一个缺省构造函数将

[1] 译注：由于p指向的内存区域本身就是从内存配置对象q中分配出来的，q可以是预先分配好的很大一个内存块，所以这里释放p指向的内存区域，回收给内存配置对象q。

会导致运行时错误，而如果不提供缺省构造函数的话，这种错误（指创建一个分配 0 或 1 个单元内存的 Pool 对象导致的错误）应该在编译时就可以检测到的。因为用户总是希望可以尽早地检测到错误，所以我们就不应该给 Pool 类提供一个缺省构造函数。因此，对于那种认为可以把缺省构造函数包括到最小标准接口之中的论断，Pool 类就是它的反例。

2.4.2　赋值运算符

假设我们给 Pool 类提供一个赋值运算符：

```
class Pool {
public:
    const Pool& operator=(const Pool& q);
    //...
};
```

赋值的正规语义（semantic）表明，这个函数把被赋的 Pool 对象转化为 Pool(n)对象，其中 n 是 q 对象分配的内存块的字节数，例如：

```
Pool p(4);        //p 分配了 4 个字节的内存块
Pool q(8);        //q 分配了 8 个字节的内存块
P=q;              //p 现在分配了 8 个字节的内存块
```

再考虑下面企图给一个 Pool 对象赋值的代码：

```
Pool p(4);
void* mem = p.alloc();
Pool q(8);
p=q;
```

在这里，用户从 Pool p 对象分配内存空间，并用 mem 储存 p.alloc 函数的返回指针，然后把 Pool q 对象赋值给 Pool p 对象。然而，Pool p 对象和 Pool q 对象分配了不同字节数的内存块，因此当用户最后想要释放 mem 指向的内存块时，

```
p.free(mem);
```

在 Pool 的任何实际实现中，混乱[1]很可能就会接踵而至。

我们可以通过给定 Pool 类赋值运算符的前提条件来避免上面这个问题：这个前提条件就是，从赋值的 Pool 对象和被赋值的 Pool 对象中分配出长度相同的内存块[2]。另一个前提条件是，我们可以要求被赋值的 Pool 对象（如 p）此时不分配任何内存块[3]。但请注意，赋值运算符的正规语义（见 2.2 节）并没有强加这些前提条件。如果我们决定强加这些前提条件，那么第二个条件的检查将会相对容易一些。因为对于第二个前提条件，我们可以通过比较 free 函数调用的次数和 alloc 函数调用的次数来获知条件是否成立；如果这两个调用次数相等，那么目前就没有已经被分配的内存块，就是说条件成立。

但为什么会如此麻烦呢？因为一个诸如 Pool 的类往往被许多 C++程序员加到他们的工

[1] 译注：因为 p 的物理内存已经发生改变，mem 现在不一定属于 P 内存区域以内了。

[2] 译注：即（如 p 和 q）都分配 mem 指向的内存块大小的内存。

[3] 译注：即 p 被赋值后是一块含有 8 个单位的内存块，没有为 mem 所指对象分配内存。

具箱里面，而且经验也显示程序员并不需要类 Pool 的赋值运算符操作。进一步说，提供 Pool::operator= 还会使那些应该在编译时检测到的错误延迟到运行时才出现。因此，类 Pool 就是把赋值运算符包含到最小标准接口的反例。

2.4.3 拷贝构造函数

现在考虑拷贝构造函数。假设我们正在设计一个类 Parser，用来描述 C++解析器。然而，Parser将会是非常复杂的——1个Parser对象要包括符号表和许多重要的内部数据结构。因此，用正确的（正规的）语义来给这样一个复杂对象实现拷贝构造函数将会是非常乏味、浪费时间和容易产生错误的。此外，用户几乎不愿意拷贝如此大的 Parser 对象。

对一个用户不需要的拷贝构造函数，如果我们花费宝贵的时间来设计、实现和测试它，那将会是毫无意义的。因此，对那种认为拷贝构造函数可以放到最小标准接口之中的论断，Parser 类就是一个很好的反例。

2.4.4 相等运算符

对某些类，定义和实现一个相等运算符会是很困难的。考虑两个 Parser 对象，即使它们的底层表述[1]具有不同的值，但它们的（抽象）值却可能是相同的（因此它们的比较结果相等）。例如，两个具有相同（抽象）值的 Parser 对象可能指向不同的符号表对象，不同的符号表对象指向不同的符号对象，而不同的符号对象又指向不同类型的对象，不同类型的对象又指向类型名不同的对象，依此类推。

由上可知，为了实现和测试一个 Parser 类的相等运算符，我们需要花费巨大的努力；然而，用户却很少想要比较 Parser 对象。因此，为 Parse 类实现相等运算符将会是毫无意义的。

2.4.5 析构函数

最后，我们来考虑析构函数。如前所述，对于任何需要析构函数而没有显式定义析构函数的类，编译器都会为它创建一个析构函数。因此，如果想找出析构函数的反例的话，我们就必须把析构函数显式声明为私有(Private)函数。

```
class T {
private:
    ~T();
    //...
};
```

（如果类的实现并不需要析构函数的话，我们就没有必要定义析构函数。）

声明了私有析构函数的类 T 将会是什么样的呢？类 T 的用户不能在静态储存器或堆栈上创建 T 对象：

[1] 译注：指实际代表的值、底层表述的值，在这里大概可以说成是类所包含的属于内置类型的成员变量的值，其他变量，直到递归转化为内置类型的成员变量为止。

```
T t;          //错误: 不能存取 private T::~T()
void f() {
    T t;      //错误: 不能存取 private T::~T()
    //...
}
```

因此，用户只能在动态存储器上创建 T 对象；但是，用户不能删除这些被创建的 T 对象：

```
    T* t = new T;
//...
delete t;   //错误: 不能存取 private T::~T()
```

希望具备这种特性（把析构函数声明为私有函数）的类是很少的，但却是实实在在存在的。假设用户运行环境具有垃圾收集器，并且由于某种原因我们要禁止手动删除 X 类型的对象，那么只要用户不需要在静态存储器和堆栈里创建 X 对象，我们就可以把类 X 的析构函数声明为私有函数。

2.5　浅拷贝和深拷贝

有两个操作，尽管它们具有某些不合乎需要的特性，但因为它们的使用范围很广，进而博得一定的注意，所以这两个操作在这里有必要特别提及一下，这两个操作就是浅拷贝操作和深拷贝操作。x 对象的浅拷贝是指：另一个和 x 相同类型的，并且它的数据成员和 x 相对应的数据成员具有相同值的对象。x 对象的深拷贝是指：另一个和 x 类型相同的对象，它具有 x 直接或间接指向的对象的一份拷贝，并且在拷贝里，所有共享和循环的联系依旧保留。考虑下面 3 个类：

```
class Z {
        // 没有数据成员
        //...
};

class Y {
private::
    Z* z;
    //没有其他数据成员
    //...
};

class X {
private:
    int i
    Y* y1;
    Y* y2;
    //没有其他数据成员
    //...
};
```

在图 2.1 中，x2 是 X 类型对象 x1 的浅拷贝，x3 是 x1 的深拷贝。

但是，对一个设计得很好，并且正确实现的程序库（几乎没有异常产生），它的用户应该可以请求创建某个对象的拷贝——通过拷贝构造函数——并且由程序库适当实现某种对象类型的拷贝。除了一些特殊的类之外，用户是不需要了解拷贝函数的实现机制的，也不应该指定用某种特殊的方式来拷贝一个对象。

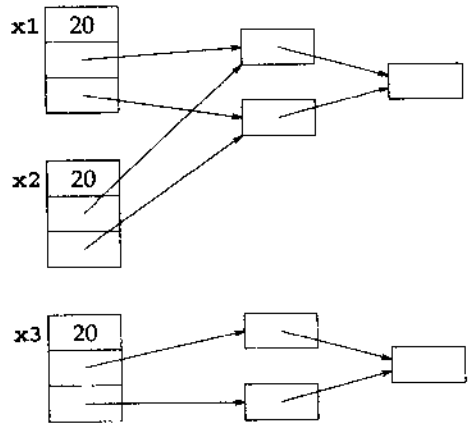


图 2.1　浅拷贝和深拷贝，x2 和 x3 分别是 x1 的浅拷贝和深拷贝

让我们还是回到讨论的话题，对一个给定的类，我们很少用浅拷贝操作或深拷贝操作来实现它的拷贝构造函数。假设我们用下面代码来实现 2.1 节中的 Rational 类：

```
class Rational {
private:
    Rational_rep* rep;
    //...
};
class Rational_rep {
private:
    int num;
    int denom;
    //...
};
```

在这里，我们只是简单地把成员变量 num 和 denom 移到一个单独的类里面。（我们将在 8.2.4 节看到，这种实现方法为类 Rational 的用户提供了链接兼容性。）下面是类 Rational 和类 Rational_rep 用深拷贝来实现的拷贝构造函数：

```
class Rational {

public:
    Rational(const Rational& r) {
        Rep = new Rational_rep(*r.rep);
    }
```

```
    //...
};
class Rational_rep {
public:
    Rational_rep(Rational_rep& rep) :
        num(rep.num),
        denom(rep.denom) {
    }
    //...
};
```

当用浅拷贝或者深拷贝来正确实现拷贝构造函数的时候，我们应该理所当然地类似（浅拷贝还有很大区别）上面那样来实现这个构造函数。但是，上面代码之所以可以实现，也仅仅是某种巧合；我们并不向用户建议这种巧合。如果类的实现改变了，那么拷贝构造函数就可能不再实现浅拷贝或者深拷贝了。实际上，用户也不应该委托拷贝构造函数实现这两种拷贝。此外，这种（巧合）现象——指通过浅拷贝或者深拷贝来实现类的拷贝构造函数——发生的概率要比程序员所认为的少很多。因为对大部分类来说，浅拷贝和深拷贝都不能用来实现类的拷贝构造函数；并且对一些特殊的类，浅拷贝和深拷贝往往带有不适合需要的属性：它们不能保持程序的不变性。例如，为了尽可能地共享类 Rational_rep，我们可以这样来改变类 Rational 的实现：

```
struct Rational_rep {
    int refcnt;     //引用计数。
    //...
};
class Rational {
public:
    Rational(const Rational& r) {
        rep = r.rep;
        ++rep->refcnt;
    }
    //...
};
```

一般当我们要共享某个对象的时候，我们必须增加引用计数，用它来决定在什么时候可以删除一个共享对象。对于任何使用类 Rational 这个版本的程序，它的不变性是指类 Rational_rep 里面的引用计数等于指向这个共享 Rational_rep 的类 Rational 的数目。读者容易看出，创建类 Rational 的浅拷贝将会违背这个不变性。

类 Rational 的问题也并不局限于浅拷贝。考虑下面的转型，它在 Rational 不为零的情况下返回真值。

```
class Rational {
public:
    operator bool() const {
            return rep->num != 0;
    }
```

```
    //...
};
```

假设用户经常调用这个函数。为了优化这个函数的实现，让我们改变类 Rational 的实现，来使所有值为零的 Rational 对象都指向同一个 Rational_rep 对象：

```
class Rational {
public:
    operator bool() const {
            return rep == rep_of_zero;
    }
    //...
private:
    static Rational_rep* rep_of_zero;
    //...
};
```

这个 bool 转型函数避免了一个间接的调用（很显然获得了一些效率的优化，但这仅仅是一个例子）。读者容易核实，创建一个值为零的 Rational 对象的深拷贝，将会破坏不变性，这里的不变性是指，所有值为零的 Rational 对象都指向同一个 Rational_rep 对象[1]。

如果浅拷贝或者深拷贝操作有可能破坏某个不变性定义的话，那么类的实现者当然可以通过插入代码以恢复这个不变性，从而避免破坏。然而，保存不变性这个做法并没有改变浅拷贝或深拷贝破坏不变性的这个事实。况且，在更加复杂的类里面，浅拷贝或者深拷贝操作也照样破坏不变性，并且这种破坏性是很难甚至不可能得到修复的（见练习 2.5c）。

假设类 X 的浅拷贝和深拷贝操作不会破坏任何不变性，那么类 X 是否应该提供这些操作呢？见下面例子：

```
class X {
public:
    X* shallow_copy();        //应该提供这个函数吗?
    X* deep_copy();           //应该提供这个函数吗?
    //...
};
```

当然不应该，除非 X 是那些非常特殊的类，并且允许用户指定它的拷贝实现方式。另一方面，如果我们想改变类 X 的实现，又不得破坏不变性，并且提供浅拷贝和深拷贝操作，那么这种改变也是不可能实现的。因此，我们可以这样认为：类一般不应该提供浅拷贝和深拷贝操作（见练习 2.5d）。

2.6 接口一致性

在类的内部和类与类之间，在程序库的内部和程序库与程序库之间，我们都应该尽可能

[1] 译注：由深拷贝的定义可知，如果执行深拷贝操作，那么也将拷贝出一个新的 Rational rep 对象，这与要求只有同一个 Rational rep 对象矛盾。

地保持类接口的一致性。很显然，保持程序库与程序库之间的接口一致性，将比保持单个程序库内部的接口一致性更加困难。

接口一致性的重要性要归因于以下几个方面。首先，具有接口一致性的类易于学习和记忆。例如，假设我们正在设计一个容器类程序库。（容器类是一种用来保存值或者对象的集合的类。）下面就是我们设计的程序库所提供的两个类：

```
template<class T>
class Set {
public:
    void insert(const T& t);
    //...
};
template<class T>
class Bag {
public:
    void insert(const T& t);
    //...
};
```

一个 Set<T>是类型为 T 的值的集合；一个 Bag<T>是类型为 T 的值的汇合。（Bag 和 Set 是有区别的，Bag 允许它里面的相同元素值出现多次，而 Set 就不允许。）尽管在 Set 里插入一个值和在 Bag 里插入一个值的区别很小（在 Bag 里插入一个已经存在的值，将会出现该值的一份拷贝，而 Set 就不会），我们还是给予这两个类的插入操作相同的句法接口（syntactic interface）。因为如果插入操作拥有不同的句法接口的话，那么对 Set 类和 Bag 类的学习、使用、记忆将会显得更加困难。

提供接口一致性的一个稍微次要的原因就是，它使用户可以更加容易地改变程序中对象的类型，假设用户已经完成了下面代码：

```
Set<int> s;
//...
s.insert(7);
```

如果写完了上面的代码之后，用户发现应该使用 Bag 而不是 Set，那么要是 Set 和 Bag 拥有一致性接口的话，这时我们就可以只更改上面的声明语句，而不需要更改调用语句：

```
Bag<int> s;          //在这里将 Set 改成 Bag...
//...
s.insert(7);         //...这一行不需要更改
```

具有相似接口的类可以从一个公共基类派生而来，譬如下面的代码：

```
template<class T>
class Container {
public:
    virtual void insert(const T& t) = 0;
    //...
};
template<class T>
```

```
class Set : public Container<T> {
public:
    virtual void insert(const T& t);
    //...
};
//类 Bag 和类 Set 类似，继承自类 Container...
```

然而，两个类仅仅具有相似的接口，并不意味着它们就应该派生自某个公共基类。假设我们的用户并不需要在 Set 和 Bag 上实现具有多态性的函数，并且如果我们希望避免由于实现为虚函数而给 insert 函数带来的额外开销，那么我们就可以决定不让类 Set 和类 Bag 派生自基类 Container。

决定两个接口的一致性程度应该经过深思熟虑。假设我们也提供一个类 Queue 来描述队列，那么类 Queue 是否应该提供 insert 操作呢？

```
template<class T>
class Queue {
public:
    void insert(const T& t);    //应该提供这个函数吗?
    //...
};
```

如果我们提供类 Queue 的 insert 操作，那么我们将在 Queue 的什么位置插入 t 呢？很显然，有两个合理的位置：队头和队尾。然而，这两个选择都是不可取的。假设选择了队头，我们还是必须提供一个在队尾进行插入的操作。另外，因为我们不能把这两个操作都命名为 insert，所以我们就必须给其中一个操作命名为别的名字：

```
template<class T>
class Queue {
public:
    void insert(const T& t);
    void insert_tail(const T& t);
    //...
};
```

然而，上面的接口就会出现内部的不一致性。许多用户往往记不清楚类 Queue 究竟是提供 insert/insert_tail 两个操作，还是提供 insert/insert_head 两个操作。这样，一些用户就会调用 insert 函数，并想当然地认为 insert 执行的是队尾插入操作。

内部一致性的接口应该给这两个操作分别命名为 insert_head 和 insert_tail:

```
template<class T>
class Queue {
public:
    void insert_head(const T& t);
    void insert_tail(const T& t);
    //...
};
```

然而，上面这个 Queue 类的接口将会与 Set 类和 Bag 类的接口不一致；为了恢复它们之

间的一致性，我们可以提供下面 3 个操作，并把 insert 定义为 insert_head 的等价操作：

```
template<class T>
class Queue {
public:
    void insert(const T& t) {insert_head(t); }
    void insert_head(const T& t);
    void insert_tail(const T& t);
    //...
};
```

很遗憾的是，这些接口比前面两种接口都要大，从而难以理解。

要让类 Queue 的接口与类 Set 和类 Bag 的接口保持一致性是很困难的，这主要是由于类 Queue 具有不同于类 Set 和类 Bag 的地方：类 Queue 逻辑上要实现两个插入操作。让我们考虑另一个像 Set 和 Bag 的类吧，它只提供一个插入操作。特别地，考虑描述堆栈的类 Stack。堆栈中只有一种插入方式，就是把插入值压入栈顶。我们是应该把这个插入操作命名为 insert 来和 Set 和 Bag 保持一致性呢？还是应该把它命名为 push 呢？因为 push 是人们对堆栈插入操作的习惯称呼；或者命名为两个呢？我们的看法是，提供一个 push 操作将使大多数类 Stack 的用户能够更容易地学习和记忆——特别地，对那些只使用类 Stack，而对我们程序库中其他的容器类并不熟悉的用户而言，就显得更加重要了。另外，并不是所有的用户都会把他们程序中的 Stack 替换成 Set 或 Bag，或者把 Set 或 Bag 替换成 Stack。因此，这种只提供 push 操作但导致不一致性的接口，在这里可能是更可取的。

接口应该尽可能地保持一致性，但也不能用一致性来代替其他的一切因素。如果一致性导致了类的接口产生对这个类不适当或者负面的影响，那么我们就不应该使用一致性。

2.7　转型

程序库设计者必须充分重视隐式转型（implicit conversion）。在 C++中，有两种方法可以用来定义从类型 From 到类型 To 的隐式转型。第一种，我们可以在类 To 中定义一个只含一个参数的构造函数（并且没有其他的缺省参数）：

```
class To {
public:
    To(const From&);    //或者是 To(From)
    //...
};
```

或者，我们可以在类 From 中定义一个转型操作：

```
class From {
public:
    operator To() const;
    //...
};
```

假如上面这两个函数中的一个（也只能是一个）存在，那么当一个类型为 From 的参数传递给需要类型为 To（或者 const To&）的参数时，就会发生隐式转型：

```
void f(To);
From from;
f(from);     //发生了隐式转型
```

2.7.1 多重所有权（Multiple Ownership）

如果 From::operator To 和 To(const From&)两个操作都已经声明了，那么对 f 的调用将会是二义性的（ambiguous）：

```
void f(To);
//...
f(from);          //如果两个转换都定义了，就会产生二义性。
f((To) from);     //仍有二义性。
```

增加一个强制转型（指(To) from）并不能消除二义性；但多重所有权问题（如此命名是由于类 From 和类 To 都拥有转型操作）是可以很容易避免的，只要 From 和 To 的设计者更加小心谨慎，不要提供两个转型操作就可以了。

对称转型的存在——就是说，一个从 From 到 To 的隐式转型和一个从 To 到 From 的隐式转型都存在——并不会导致相同的二义性问题。

```
void f(To);
void g(From);
To to;
From from;
f(from);     //发生 From 到 To 的隐式转型
g(to);       //发生 To 到 From 地隐式转型
```

实际上，现实的类一般都提供对称转型。考虑一个用于表述正则表达式[SM77]的类 Regex，因为有必要用一个字符串来构造一个 Regex 对象，并且有必要把一个正则表达式解释成一个字符串，所以一个真实的 Regex 类提供了和 String 类的对称转型：

```
class Regex {
public:
    Regex(const String& s);
    Operator String() cosnt;
    //...
};
```

如果不能改变类 String 的定义，那么我们就只能都在类 Regex 内定义这两个转型操作了。

2.7.2 敏感转型

一个从 From 到 To 的隐式转型，如果它表述的是一个从 From 到 To 的自然映射，或者是用户想要默许发生的转型，我们就说这个转型是敏感的（sensible）。实际上，大多数转型都应该是敏感的（我们会在下一节讨论特殊情况）。

下面让我们来考虑几个有关敏感性（sensibility）的例子。假设我们是一个数学程序库的

设计者，在我们的程序库里，类 Rational 和类 Complex 分别描述有理数和复数。假如我们只考虑 int、double、Rational 和 Complex 4 种类型，就会有 12 种可能的隐式转型；那么，这些转型中哪些是敏感的呢？

- Complex 到 int 的转型

很显然，这个转型不是敏感的（sensible）。尽管我们有可能定义任何从复数到整数的映射，但这种定义肯定不是自然的。

- int 到 Complex 的转型

这是一个很明显的映射：从整数 x 映射到复数 x+(0)($\sqrt{-1}$)，并且，用户大多数都希望整数可以默认地转化为复数。因此，这个转型是敏感的。

- Rational 到 double 的转型

将一个 Rational 对象映射到一个 double 对象，这个 double 对象储存的值是 Rational 对象的分子除以分母得到的近似值，整个过程将可能损失一定的精度。尽管这个转型有时可以被认为是自然的，但是用户往往不想让这种转型默认地进行（因为精度可能损失）。因此，这种转型不是敏感的。

- double 到 Rational 的转型

如果我们认为 double 描述的是实数集合，那么从 double 到 Rational 将没有自然的映射；但实际上，每个 double 对象表述的是一个有限小数，并且有限小数和有理数有着很好的自然映射，因此，这个转型有可能是敏感的。

- Complex 到 Rational 的转型

由于不存在用户希望发生的、从 Complex 到 Rational 的自然映射，所以这个转型不是敏感的。

- Rational 到 Complex 的转型

这个转型是比较复杂的。每一个有理数同时也可以是一个复数，因此我们可以认为存在一个从 Rational 到 Complex 的映射。然而，如果 Complex 只能以 x+(y)($\sqrt{-1}$)的形式来表示复数，其中 x 和 y 都是 double 类型，那么这个转型就不是敏感的了，原因和 Rational 到 double 的转型不是敏感的原因相同（即精度损失）。

从上面这些例子可以看出，经过了详细的分析，很多隐式转型都不是敏感的，因此，我们就不应该提供这类转型操作。练习 2.6 要求读者判断剩余转型的敏感性。

注意，对于某些非敏感的隐式转型，如果我们的程序确实需要它所实现的功能，那么我们可以把这种转型实现为显式转型。例如，尽管从 Rational 到 double 的隐式转型不是敏感的，并且会有精度的损失，但用户却可以显式地（explicitly）将 Rational 对象转型为 double 对象；我们可以在数学程序库里提供这个函数：

```
class Rational {
public:
    double to_double() const;
    //...
```

```
};
```

2.7.3 不敏感转型

回想一下2.4.1节的Pool类，如2.4.1节所述，Pool的构造函数需要包含一个大小参数：

```
Pool::Pool(size_t n) {/* ... */};
```

此外，我们也没有给这个函数传递其他的参数。因此，我们会得到一个单参数的构造函数，并由它来创建一个转型函数。遗憾的是，从size_t到Pool的转型并不是敏感的——几乎没有用户会把一个size_t对象默认地转化为一个Pool对象：

```
void f(const Pool& p);
//...
f(17);  //合法，但无疑是错误的[1]
```

我们可以用两种方法来解决上面这个问题。第一种，我们可以只提供构造函数，把避免隐式转型的工作留给用户完成；第二种，我们可以定义一个中间类：

```
class Pool {
public:
    class Size {
    public:
        Size(size_t n);
        //...
    };
    Pool(Size n);
    //...
};
```

因为对函数的实参，如果它的转型是用户定义的，那么C++是不能（隐式地）执行多于一次的转型的；这个设计将会导致编译器拒绝编译下面的错误代码：

```
void f(const Pool&);
//...
f(17);  //错误
```

然而，定义一个中间类有很大的缺点：它使Pool类的理解和使用更加困难。现在如果想要构造一个Pool对象，用户应该这样编码：

```
Pool p(Pool::Size(17));
```

大多数用户都会倾向于使用虽容易产生错误，但更加简单的接口；因此，程序库有时也提供不敏感的（nonsensible）转型。

2.7.4 转型数目（Fanout）

我们可以这样来定义一个类型的转型数目（fanout）：它就是这个类型可以隐式转换为其他类型的数目。很大的转型数目往往不是我们所期望的，因为它很容易导致二义性的发生。例如，假设类From可以转型为两种类型：To和Another_to，那么下面的函数调用就存在二

[1] 译注：实际上C++标准中的一个新关键字explicit已经很好地解决了这个问题。

义性：

```
void f(To);
void f(Another_to);
From from;
f(from);          //二义性
```

增加一个强制转型（cast）就可以解决这个二义性问题：

```
f((To) from);   //OK,发生了从 From 到 To 的转型
```

然而，如果可能的话，C++程序库不应该强迫它的用户在他们的代码中使用强制转型（cast）。因此，C++程序库应该避免大的转型数目。幸运的是，只提供敏感转型的程序库一般只具有小的转型数目，特殊情况就是一些诸如 int 和 char*的内建类型；因为很多类定义了具有这种类型单参数的构造函数，于是 int 和 char*就具有很大的转型数目。为了避免二义性问题，程序库用户无论在什么时候，都应该避免依赖从内建类型到程序库定义类型之间的隐式转型。

2.8　const 关键字的使用

在程序库中，const 关键字的正确使用是很重要的。使用 const 的最大障碍就是用户往往未能正确理解 const 的意义。接下来，我们将讨论如何解释 const，如何使用 const，和当我们要改变 const 的时候，const 为什么不能够被重新解释。

2.8.1　抽象 const 对比位元 const

我们可以用好几种方式来解释关键字 const。先考虑函数 sqrt，它用于计算 Rational 对象的平方根（这里的 Rational 指 2.1 节描述有理数的类）：

```
Rational sqrt(const Rational& r);
```

如果我们采用抽象解释方式，那么上面这个声明语句说明，sqrt 函数不会使用 r 去改变 r 所引用对象的抽象值。如果我们采用位元解释方式，那么这个声明语句说明，sqrt 函数不会使用 r 去改变组成 r 所引用对象的任何位元。（练习 2.10 讨论了其他几种可能的解释方式。）

相对于抽象 const，位元 const 有一个优点和几个缺点。如果在任何地方都使用位元 const，那么，对于没有构造函数和析构函数类型的 const 对象，我们就可以把它安全地储存在只读内存区域（ROM, read-only memory）。而且，对一些应用程序而言，把对象储存在只读内存区域将会是一个很重要的优化方式。然而，位元 const 也具有一些缺点：它具有比抽象 const 更低级别的抽象性。实际上，一个 C++程序库接口的抽象性级别越低，使用这个程序库就越困难。

而且，使用位元 const 的程序库接口暴露了程序库的实现细节。任何时候的实现细节都被暴露无遗了，而这往往会带来一些负面效应。例如，假设在我们程序库的最新版本中，我们决定这样来优化 sqrt 函数：使类 r 的 num（分子）和 denom（分母）数据成员转化成最简形式。如果采用的是位元 const 的方式，这时对 sqrt 函数实现的改变将要求我们去除 sqrt 声

明语句的 const 关键字，而这违反了我们原先使用 const 的本意。更加遗憾的是，这个改变是源代码不兼容的，因此可能会破坏用户的代码。

因此，在程序库接口和程序库实现细节上面，程序库设计者都应该使用抽象 const。考虑下面的代码：

```
class Rational {
    //...
private:
    void reduce() const;
    int num;
    int denom;
};
```

函数 reduce 把 num 和 denom 转变成最简形式。因为简化有理数的表示并没有改变这个有理数的（抽象）值，所以我们把 reduce 定义为 const 函数。

现在考虑下面 reduce 函数的实现代码：

```
void Rational::reduce() const {
    int gcd = GCD(num,denom);
    num /= gcd;      //错误，不能改变 *this，因为 const 的限制
    demon  = gcd;    //错误，不能改变 *this，因为 const 的限制
```

在这里，GCD 是一个返回它的两个参数的最大公约数的函数。遗憾的是，试图改变 num 和 denom 的语句是非法的，因为 C++编译器根本没有办法知道这样的事实：对 num 和 denom 都进行化简并没有改变 this 指针指向对象的值。

我们可以用 3 种办法来解决这个问题。首先，我们可以把 num 和 denom 声明为 mutable（可变的）：

```
class Rational {
private:
    mutable int num;
    mutable int denom;
    //...
};
```

现在，任何试图对 const Rational 的 num 和 denom 成员变量的改变都是合法的（包括试图改变它们中的一个或者两个，从而导致 Rational 对象值的改变的操作，都是合法的）。另一种允许改变 num 和 denom 的技术是：只在我们需要改变的地方，去除 const(cast away const)：

```
void Rational::reduce() const {
    //...
    const_cast<int> (num) /= gcd;        //ok
    const_cast<int> (denom) /= gcd;      //ok
}
```

由于 mutable 和 const_cast 是 C++相对较新的特性，因此现今有些编译器并不能实现它们；从而，使用它们的代码有时就不具备可移植性。另外，一个具有更好可移植性的技术是使用旧式的强制转型：

```
void Rational::reduce() const {
    //...
    Rational* let_me_modify = (Rational*) this;
    let_me_modify >num /= gcd;      //ok
    let_me_modify->denom /=gcd;     //ok
}
```

试图使用旧式的 cast 转型对 const 对应的 X 对象进行转型，只有当被转型的类 X 具有不少于一个的显示构造函数时，类 X 的转型后的对象才会有 X 原来定义的行为[1]。(大多数重要的类都能满足这个限制条件，即至少有一个显式构造函数)

第三种避免编译器产生错误的方法是：对我们要改变的对象增加一个间接层(indirection)：

```
class Rational {
public:
    Rational() : num(new int), denom(new int) {/* ... */}
    //...
private:
    int* num;
    int* denom;
};
```

那么，改变*num 和*denom 是合法的，即使在 Rational 的成员函数里面也是如此：

```
void Rational::reduce() const {
    //...
    *num /= gcd;     //ok,因为 Rational 的成员变量 num 并没有改变，它还是指向
                     //原来的地址。
    *denom /= gcd;   //ok，同上
}
```

然而，增加一个间接层降低了程序的效率。因此，使用关键字 mutable 将是解决这个问题最好的办法，除非可移植性是主要的关注因素。

2.8.2　最大限度地使用 const

许多 C++程序员只把 const 当成一个不能捕获错误的讨厌东西，因此，并不是所有的程序员都能充分地使用 const。然而，作为 C++程序库的设计者，就没有这么多是否使用 const 的自由了。对于 C++程序库的接口，应该在它应用的每个地方都使用 const 关键字——就是说，使用 const 的每个地方都可以确保程序库在此处不会被修改。

如果未能最大限度地使用 const，那么很有可能会给程序库用户带来问题。假设我们想要提供一个库函数，它带有两个参数——1 个指向以 null 结束的字符串指针 p 和一个指针数组 a，a 的元素也是以 null 结束的字符串指针——并且，如果 p 指向的字符串和 a 中某个指针元素指向的字符串相等，就返回真值。我们很可能会像下面这样定义这个函数：

[1] 译注：如上例，是指只有当 Rational 具有公共构造函数时，转型后的对象 Let_me_modify 才能调用它的成员变量 num 和 denom。

```
//没有最大限度使用 const
bool contains(const char** a, const char* p);
```

对编写下面代码的用户而言，这个接口可以顺利通过编译：

```
static const char* keyword[] = {
    "array", "of", "four", "strings"
};
bool iskeyword(const char* p) {
    return contains(keyword, p);
}
```

然而，下面的代码却不能通过编译：

```
bool iskeyword(const char* const* keyword, const char* p)
{
    return contains(keywords, p);        //错误
}
```

这个错误来源于我们的失误，因为我们没有在每个应该使用 const 的地方都使用 const，下面是 contains 正确的接口：

```
//最大限度地使用 const
bool contains(const char* const* a, const char* p);
```

现在所有的用户代码，不管是充分使用 const 的代码，还是没有使用 const 的代码，都可以如用户预期地通过编译（或者是不通过编译，但也是用户预期的）。

对最大限度地使用 const 的规律存在着一个例外：假设 contains 函数并没有改变它的参数 a 和 p 的值，我们仍然不应该如下声明 contains 函数：

```
//不好的想法
bool contains(const char* const* const a,        //增加了一个 const
    const char* const p);                        //增加了一个 const
```

我们在这里增加的 const 对用户没有产生任何影响，并且，用户也很少被非引用的（nonreference）参数所影响。而且，如果 contains 将来的版本由于某种原因需要改变参数 a 或 p 的值，那么这些相应的 const 也应该被删去，这将破坏兼容性。因此，const 决不能用于改变非引用（nonreference）参数的值。

2.8.3 对 const 不安全的解释

有时，程序库设计者希望能对 const 作不同于抽象解释和位元解释的另一种解释，然而，大多数对 const 的解释并不是类型安全的。考虑下面的类：

```
class Noderef {
public:
    int value() const;
    void setvalue(int val) const;
    const Noderef& operator=(const Noderef& n);
    //...
};
```

Noderef 是一个指向底层节点的引用；每一个底层节点都只包含一个 int 值。函数 value

返回节点储存的这个 int 值，而函数 setvalue 则把 val 的值赋给底层节点这个值。由上面代码可以看出，value 函数和 setvalue 函数都是 const 函数，因此这两个函数都不会改变 Noderef 本身的值（指引用值，类似指针值，而并不是节点值）。

把 setvalue 成员函数声明为 const 函数多少会让人有些惊讶，因此类 Noderef 的设计者可能希望在应用 Noderef 的时候，可以重新解释 const。例如，设计者可能如下重新解释所有 const 的用法：

储存在底层节点的值并没有改变。

然而，这个对 const 的重新解释是不安全的。下面是在所提出的重新解释下类 Noderef 的声明：

```
//具有对 const 不安全重新解释的类 Noderef
class Noderef {
public:
    int value() const;
    void setvalue(int val);
    const Noderef& operator=(const Noderef& n) const;
    //...
};
```

在上面的代码中，setvalue 已经不是 const 函数了，但现在赋值运算符变成了 const 函数！（赋值运算符改变了类 Noderef 中的值，但不是储存在底层节点的值。）然而，这个接口（setvalue 函数）包含了一个类型漏洞，请考虑下面的代码：

```
void f(const Noderef& n) {
    Noderef m = n;      //m 和 n 现在引用（指向）相同的节点
    m.setvalue(0);      //oops!
}
```

函数 f 在它的声明中，保证 f 将不会使用 n 来改变 n 所引用节点的值，但在我们对 const 的重新解释下，使用 m 对函数 setvalue 的调用却违反了这个保证，并且还可以顺利通过编译，而不产生任何错误或警告。而且，读者可以证明：为了避免这个类型漏洞，最简单的权宜之计莫过于，删去 Noderef 赋值运算符声明语句中的最后一个 const，但这并不能成功，也达不到我们的目的。

因此，我们应该尽量避免对 const 进行重新解释。

2.9　总结

正规函数——拷贝构造函数、析构函数、基本赋值运算符、相等运算符和不等运算符——在所有的类中都应该实现相同的语义。

尽管没有最小标准接口，但是 nice 函数——缺省构造函数、拷贝构造函数、赋值运算符和相等运算符——应该是大多数类都提供的函数。没有任何函数是所有的类都应该提供的函数；而且，绝大多数类都不应该提供浅拷贝和深拷贝操作。

对程序库中类的接口一致性，我们应该给予充分的重视。但是当一致性使类的接口变得很不适当或者不直观时，我们就不能一味顽固地坚持这种一致性。

在决定提供何种类型的转型时，程序库的设计者应该提供敏感的并不会导致多重所有权的转型；而尽可能避免提供不敏感(nonsensible)的转型，并且限制转型数目。

程序库中 const 关键字的使用也要引起充分的注意。一般说来，程序库应该在它们的接口实现抽象的 const，并且在每个程序库需要的地方都使用 const 关键字。

2.10 练习

2.1 给出下面被建议为最小标准接口函数的反例：

a. 输入函数；

b. 输出函数；

c. 用字符串返回外层类类名的函数。

2.2 考虑类 WORM_Pool，它和 2.4.1 节的 Pool 类很相似，但这一点除外，它在只能写一次但可读多次的内存区域分配内存块。那么，类 WORD_Pool 是析构函数的反例吗？请说明是或不是的原因。

2.3 假设我们为用户提供一个类 Buf，它描述一个缓冲区：

```
class Buf {
public:
    Buf(size_t sz);
    //...
};
```

类 Buf 的构造函数创建了一个 sz 个字符大小的缓冲区。假设以后我们的用户想要传递指向 Buf 的指针，并且他们可以操纵这些指向 Buf 的指针。这样就有很多指针指向 Buf，以致于他们难以决定何时可以安全地删除 Buf。

a. 为了帮助我们的用户，我们可以创建另外一个类 Bufptr，它用来描述一个指向 Buf 的智能指针：

```
class Bufptr {
public:
    Bufptr(Buf* p);
    Buf* operator->() {return rep;}
    //...
private:
    Buf* rep;
};
```

Bufptr 的构造函数创建了一个指向 p 的智能指针，operator->返回这个指针。我们也可以在 Buf 中增加一个指向 Buf 的引用计数：

```
class Buf {
```

```
private:
    friend class Bufptr;
    int refcnt;
    //...
};
```

refcnt 的值总等于指向 Buf 的 Bufptr 的数目，并且，类 Bufptr 的各种成员函数会维护 refptr 的值。当没有其他 Bufptr 对象指向这个 Buf 的时候，最后一个 Bufptr 对象的析构函数会删除它所指向的 Buf。那么，如果我们把类 Buf 的析构函数声明为私有函数，我们应该防止哪些在用户端可能会出现的错误呢？把类 Buf 的析构函数声明为私有函数的缺点又是什么呢？

b．假设我们没有提供类 Buf 和类 Bufptr，而是提供一个单独的类 Bufref，它描述一个指向底层缓冲区的引用：

```
class Bufref {
    public:
        Bufref(size_t sz);
        Bufref(const Bufref& b);
        //...
    };
```

它的构造函数将创建一个引用，这个引用指向一块新分配 sz 个字符的底层缓冲区。拷贝构造函数（具有正规语义）也创建一个新的引用，但它指向 b 所指向的已经存在的缓冲区。Bufref 还提供了一个析构函数，如果已经没有其他的 Bufref 对象指向这块缓冲区的话，这个析构函数将会删除这块底层缓冲区。请说出类 Bufptr 相对于类 Buf 和类 Bufptr 的优点在什么地方？

c．对于析构函数作为最小标准接口中的函数这个问题，类 Buf 和类 Bufptr 是否可以作为令人信服的反例？

2.4　在 2.4 节里，我们给出了一个类可以作为最小标准接口函数的反例的 3 个原因，在这个练习里，我们还给出另外两个理论上的但在实际中并不会出现的原因。

a．Complexity_class 是一个模拟复杂类型([HU79])的类，假设我们可以创建 Complexity_class 的实例来模拟复杂类型 P 和复杂类型 NP。那么，针对提议的最小标准接口函数，Complexity_class 可以作为哪些函数的反例呢？

b．(*) 考虑两个类——TM 描述一台图灵机[DW83]，Tmset 表示图灵机的集合。假设我们可以创建 Tmset 的实例，用它来表述图灵机的集合，并且这些图灵机在空白磁带开始处将会停止运转。进一步假设存在下面的函数，它将返回 tms 和{tm}的联合：

```
Tmset operator+(const Tmset& tms, const TM& tm);
```

请证明（或说明）：Tmset 的相等运算符（operator==）的实现将要求计算一个不可计算的函数（incomputable function）。

2.5　这个练习用于进一步考察浅拷贝和深拷贝。

a．给出一个现实存在的类，它用浅拷贝操作实现它的拷贝构造函数。

b．(*) 给出一个现实存在的类，它至少拥有一个指针数据成员，指针的实现不存在内存

泄漏，并且通过浅拷贝操作实现它的拷贝构造函数。

c. (**)给出一个现实存在的类，它的浅拷贝操作会破坏某种不变性（invariant,见 2.5 节），并且这种破坏性是不可以修复的。给出一个现实存在的类，它的深拷贝操作会破坏某种不变性，并且这种破坏性也是不可以修复的。

d. 给出一个现实存在的类的实例，它拥有一个强参数，这个强参数用于提供深拷贝和浅拷贝操作。

2.6 在下面的 int、double 和 2.7 节中的 Rational、Complex4 个类型之间，哪些转型是敏感的（sensible）：

- Rational 到 int 的转型；
- int 到 Rational 的转型；
- complex 到 double 的转型；
- double 到 Complex 的转型。

2.7 C++中的哪些内建的算法转型是敏感的？

2.8 编写一个和 C++内建类型 int 相似的类 Int 将是相当困难的。

a. 编写 Int 类的接口函数（就是说，给出 Int 类的声明），切记要提供适当的转型函数。

b. (*) 尽管你尽了最大努力，但你设计的类 Int 的行为和内建 int 类型的行为有哪些差别？

c. 假设你同时需要提供类 Char、Short、Long、Float 和 Double，那么你的 Int 类需要哪些其他的转型操作呢？Char、Short、Long、Float 和 Double 中的那些类需要具有到 Int 类的转型操作吗？

2.9 克林法则（Kleen's theorem，见[DW83]）认为一种语言当且仅当它可以表示为正则表达式的时候，才能被有限状态的接受者所接受。

a. (*) 假设你的程序库不但提供了一个类 FSA(finite state acceptor)，用它模拟有限状态的接受者；还提供了另一个类 Regex，用 Regex 来模拟正则表达式。这时，如果要在你的程序库模拟克林法则的话，应该提供什么样的类和函数呢？

b. 如果你的解决方法用到了任何隐式转型，并且只有敏感的转型的话，请给出避免多重所有权(multiple ownership)和不必要的转型数目(fanout)的方法。

c. 假设你的程序库提供了类 Regex，但没有提供类 FSA，而你的某些用户使用了一个提供类 FSA 的程序库；为了使你的用户能够容易地使用 a 部分所提供的相同功能，应该如何设计你的程序库呢？这个设计的缺点又是什么？

2.10 考虑 2.8.3 节的 Noderef 类.

a. 当用 const 来修饰 Noderef 的时候，如果我们是这样来解释 const：既不改变 Noderef 的值，也不改变底层节点的储存值，那么将会有什么问题发生？

b. 我们是否可以这样来设计 Noderef：它的接口禁止用户改变底层节点的存储值？如果

可以，应该如何设计？

2.11　假设我们希望提供一个函数 firstvowel，它返回一个指向给定字符串中的第一个元音字母的指针，这个字符串以 null 为结束字符；如果这个字符串没有元音字母，则返回 0。考虑下列建议的接口：

```
char* firstvowel(char* s);                    //  1
const char* firstvowel(char* s);              //  2
char* firstvowel(const char* s);              //  3
const char* firstvowel(const char* s);        //  4
```

a. 对于（1）到（4）的每个接口，如果我们只给用户提供其中的一个接口，请分别说出会发生什么问题？

b. 对于在 a 部分中发现的问题，请给出你的解决方法？

2.11　参考文献和相关资料

Cargill [Car92]、Cline 和 Lomow [CL95]、Barton 和 Nackman [BN94]都讨论了一些关于如何设计好的类的话题。

Liskow 和 Guttag [LG86]讨论了一些关于抽象状态的概念和其他的一些抽象法则的问题。

正规语义（regular semantics）的术语虽然是很新的，但它所描述的原则已经被好的程序员坚持了很多年了。nice 类的第一次使用出现在 Lee 和 Stepanov 的[LS93]中，它和本文的 nice 类在意思上略有差异。

在继承面前正确地实现赋值运算符是相当棘手的，更多信息请参阅 *C++ Report* 杂志中的 Meyers 文档 [Mey94c, Mey94a, Mey94b]。

Meyers [Mey92c]也给出了对最小标准接口建议的批判。

这章里的 Pool 类是由 Koenig 在[UNI92]里设计的一个类改编而来的。

Doug Lea 建议在 2.4.5 节里的例子使用垃圾收集机制。

浅拷贝和深拷贝操作导致的问题并不仅仅局限于 C++语言，在所有的编程语言都会出现。Knight[Knig93]讨论了这种操作在 Smalltalk 语言中导致的问题。Gorlen, Orlow 和 Plexico 在[GOP90]中给出了一种在 C++程序库实现浅拷贝和深拷贝的技术。

Murray 在[Mur88]里杜撰了多重所有权问题（multiple ownership problem）的术语。

要想设计一个可以精确模拟 C++真实指针行为的 C++智能指针是不可能的，Edelson 在[Ede92]解释了这个原因。

第3章 扩展性

在你对一个类库进行扩展之前，请不要说你已经完全了解了这个类库。
——Johanna Kaspar Lavater

对于许多C++程序库，扩展性是一个非常重要的特性。在这一章里，我们将会看到关于是否提供扩展性的一些权衡之计。对一个C++程序库，实现扩展性最普遍的方法就是从它的类派生出新的子类；然而，如果要实现继承，程序库里每一个有用的基类都有必要对它的派生类给予一定的限制。之后，我们定义了函数继承语义（semantic）的概念。因为对于类里面的所有函数，如果能够清晰定义它们的继承语义，那么将会提高这个类的可继承性。接下来，我们列举了几个扩展性最常见的障碍，并给出了克服这些障碍的技术。我们还详细说明了一个在继承中经常遇到的问题——派生赋值问题——并且给出了如何在程序库中避免这个问题的方法。最后，我们讨论了给用户提供程序库源代码的优点，这样，当从程序库进行派生并遇到障碍的时候，用户就可以自己克服这些障碍。

3.1 扩展性的权衡

某个事物如果是可以进行延伸扩展的，我们就说它具有扩展性——这就是说，它的范围、含义或者应用是可以增加的。例如，下列事物是可扩展的：

- 生活语言的词汇；
- 法语的合法字符；
- C++程序里类型的集合。

生活语言的词汇是可以扩展的——一个词汇可以摘录自一本科幻小说（如robot），技术进步也给语言引进了一些新的词汇（如television和camcorder），新的词汇还可以通过合并旧

词汇而产生（如 smog 就是通过合并 smoke 和 fog 而产生；motel 就是 motor 和 hotel 的合成词）。创造新词的方法很多，不一而足。法语的合法字符也是可以按照指定的程序（这里的程序不是计算机程序，指一个过程）扩展的。对一个 C++程序，只要我们再定义一个新的类和这个类相应的操作，它的类型集合也是可以扩展的。

任何情况下都可以扩展的事物是不存在的。例如，一个音乐盒的制作者，他可能希望他制作的音乐盒可以让用户自由增加播放的乐曲。这可以通过给音乐盒插入一个组件来实现，根据插入组件的差异，音乐盒就会演奏不同的乐曲。然而，这个音乐盒往往只能接受某一特定制作商的组件，而对其他制造商的组件就不能使用了。

扩展性设计处处都要求权衡优缺点。可扩展的音乐盒的制作将比只播放单一乐曲（如 Johannes Brahms 的“Lullaby”）的音乐盒的制作更加困难。因此，相比不可扩展的音乐盒，可扩展音乐盒的制作成本将会更高。

另外，产品的用户往往不需要产品具有扩展性，因此他们也就不愿意承受因扩展性而带来的额外成本。假设我们的音乐盒制作者预先知道用户除了播放乐曲“Lullaby”之外，只会播放另一首乐曲“Bicycle Built for Two”，那么，与其制作一个可扩展的音乐盒，不如把这两首乐曲都放进音乐盒里面，并提供一个可以在两首乐曲中进行选择的开关，让用户可以选择喜爱的乐曲。比起只有一首乐曲的音乐盒，这样（两首乐曲）的音乐盒将具有更好的扩展性；但就扩展性而言，它或许比不上可扩展（可以插入组件选择乐曲）的音乐盒。

上面的例子说明，有时我们可以避免因产品的扩展性而带来的成本。假设我们的产品只需要很少的扩展性，并且我们已经预先知道这些扩展功能，那么，与其提供一个可扩展的产品，还不如提供一个包含所有已知扩展功能的产品，而且这些扩展功能可以很好地满足用户的需要。

3.2 扩展性和继承

继承是 C++程序库用来提供扩展性的主要机制。有时，从一个类实现继承是容易的；但某些时候，从一个类继承却是很难实现的。类派生的困难程度取决于派生类本身与基类的设计与实现。对于一个类而言，如果可以容易地从它派生出适当的派生类，我们就称这个类是可继承的。之所以说是适当的派生类，因为没有人会这样认为：由于不能从类 Military_vehicle（军用交通工具）派生出类 Washing_machine（洗衣机），所以类 Military_vehicle 就不具备继承性。但当类 Military_vehicle 不能容易地派生出类 Jeep（吉普车）的时候，我们就可以认为它缺乏继承性。即我们考虑的继承，是基类和派生类是同属于某个小范畴里面的。

为了理解一个类具有继承性所需要的品质和特性，先考虑下面继承的 3 个用途：

- 用户只希望继承基类的接口，而并不继承基类的实现。接口类将提供这种继承的实现，我们将在 3.2.1 小节讨论接口类。
- 用户只希望继承基类的实现，而并不继承基类的接口。私有派生将实现这种继承。

- 用户既想继承基类的接口，又想继承基类的实现。从非接口类中公共派生出子类将实现这种继承。

(你或许看到我们并没有讨论保护（protected）派生，这是因为对于保护派生有用与否的观点是有争论的。)

3.2.1　只继承基类的接口

如果希望用户只继承基类的接口，而不继承基类的实现，我们就可以提供一个接口类——接口类是指这样的类：不包含数据成员，所有的成员函数都是纯虚函数，并且它的所有基类都是接口类。

例如，假设我们需要提供一个二分查找树——指存储在每一个节点 n 的值都比存储在它左子树任何节点上的值大，而比存储在它右子树任何节点上的值小[CLR90]。我们可以用典型的方法来实现二分查找树类 BSTree——特别地，树中的每一个节点用一个对象来表示，每个节点对象包含两个分别指向它的左子节点和右子节点的指针。假设在这之后，我们的用户希望创建一个类 Complete_bstree，用来描述满二分查找树（在满二分查找树里，包含节点的每一层都填满了节点的，节点的层是指此节点到树的根节点的距离。）于是，对于类 Complete_bstree，与其用子节点指针指向的节点来实现，还不如使用压缩数组来实现[CLR90]。所以，只有对 BSTree 类进行重新实现，才能使它对类 Complete_bstree 也是可继承的。

为了避免(需要重新实现)这个缺点，我们可以让类 Bstree 继承自一个接口类 BSTree_int:

```
template<class T>
class BSTree_int {
public:
    virtual void insert(const T& t) = 0;
    virtual bool contains(const T& t) = 0;
    //…
    //只有纯虚函数，没有别的成员函数
    //...
};
template<class T>
class BSTree : public virtual BSTree_int<T> {
public:
    void insert(const T& t);
    bool contains(const T& t);
};
```

模板参数 T 的类型就是树中储存在每个节点中的值的类型。而且，在上面代码中，我们可以从类 BSTree_int 虚拟派生出其他的类（如 BSTree），这是因为当使用接口类的时候，它可以有很多条派生路径，从而来派生出一些不同的子类。

由于接口类 BSTree_int 并没有包含类的实现，所以我们的用户就可以很容易地从它派生出类 Complete_bstree，并且使用压缩数组来描述类 Complete_bstree：

```
template<class T>
```

```
class Complete_bstree : public virtual BSTree_int<T> {
    //...
};
```

对于程序库提供的任何函数，如果这些函数使用了指向二分查找树的指针或者引用，那么就应该尽可能地使用指向类 BSTree_int 的指针或引用，而不是指向类 BSTree 的指针或引用。这个设计方法将令用户可以传递Complete_bstree对象给这些函数(因为类Complete_bstree也是类 BSTree_int 的派生类)。对于声明要接收指向 X 直接基类或间接基类的指针或引用为参数的函数，我们可以传递一个指向 X 类型的对象的指针或引用给这个函数，来作为这个函数的参数，这种能力叫做可置换性。

如果类 X 是接口类，或者类 X 中的所有公共成员函数都在至少一个类 X 的直接或间接基类中声明过，那么就可以把类 X 看作接口类。为了提高程序库的扩展性，我们可以接口化程序库中的每一个类。所以，对于以扩展性为首要设计目标的程序库，这种把每个类都设计为接口类的设计方法是非常可取的，我们将在 8.2.2 小节进一步讨论这种方法。

3.2.2 只继承基类的实现

用户有时只希望继承基类的实现，而并不打算用派生类对象来代替基类的类型对象，那么私有继承将是一种很好的实现方法。例如，假设我们的用户希望用类 BSTree 来实现一个 Map 类，其中 Map<X,Y>表示从 X 类型的值到 Y 类型值的映射（Map 有时也称为关联数组或者字典）。类 Map 的接口可能如下所示：

```
template<class X, class Y>
class Map {
public:
        void insert(const X& x, const Y& y);
        void remove(const X& x);
        bool contains(const X& x);
        Y valueat(const X& x);
};
```

函数 insert 在删除了所有原来第一个值为 x 的 Map<X, Y>映射对之后，添加<x,y>映射对到 Map 集合中。函数 remove 删除所有第一个值为 x 的 Map<X, Y>映射对。当在 Map 集合中含有第一个值为 x 的 Map<X, Y>对时，函数 constains 将返回 true 值，否则返回 false 值；valueat 返回第一个值为 x 的 Map<X, Y>对中对应的第二个值（即 y 值）。

我们的目的是能够通过类 BSTree 来实现类 Map，假设我们在类 BSTree 里面提供了一些虚函数，这样 Map 的实现者就可以改写这些虚函数了（参考练习 3.2）。另一方面，由于映射并不属于二分查找树中的一种，所以用户也就很难用 Map 类型的对象来置换 BSTree 类型的对象。因此，为了能够通过 BSTree 类来实现 Map 类，我们往往采取私有继承方式：

```
template<class X, class Y>
class Map : private BSTree<X> {
    //改写我们在类 BSTree 中提供的(虚)函数
    //...
```

```
};
```

练习 3.3 还给出了一种通过类 BSTree，但不使用私有继承实现类 Map 的方法，这也是一种稍微更加困难的实现方法。

3.2.3 同时继承基类的接口和实现

用户常常希望同时继承基类的接口和实现。假设我们的用户希望创建一个类 AVLTree，用来表述 AVL 树(AVL 树是一种具有如下性质的二分查找树：对于树中的任一个节点 n，它的左子树中沿最长路径从节点 n 到叶子节点经过的节点个数，和它的右子树中沿最长路径从节点 n 到叶子节点经过的节点个数相差不超过 1 个（即二叉平衡树）[AHU83])。由于 AVL 树是一种二分查找树，所以用户希望类 AVLTree 能够继承类 BSTree 的接口；另一方面，由于类 AVLTree 的实现和类 BSTree 的实现很相似，用户也希望类 AVLTree 能够继承类 BSTree 的实现。为了既继承类的接口，又继承类的实现，我们就应该从非接口类 BSTree 公共派生出类 AVLTree：

```
template<class T>
class AVLTree : public BSTree<T> {
    //...
};
```

当然，为了使上面的这种公共继承能够顺利实现，类 BSTree 必须是一个可继承的基类。

3.3 继承语义（Semantic）

类继承语义是指类本身应该满足的语义和期望它所有的公共派生类（无论是直接派生还是间接派生）都应该满足的语义。假设类 D 直接公共派生自基类 B，为了使这种派生关系能够正常生效，类的实现者和类的使用者都希望类 B 和类 D 符合下面的法则：

- 类 B 的实现者必须提供有关 B 的继承语义的文档资料。
- 所有类 B 的用户（包括 B 的实现者）在操纵指向 B 的指针或者引用时，都必须假定 B 的继承语义约束是存在的。
- 类 D 的实现者必须确保类 D 符合类 B 的继承语义。
- 类 D 的实现者必须定义类 D 的继承语义，于是对于类 D 的所有派生类，只要满足类 D 的继承语义，也就满足类 B 的继承语义。

首先，假设我们的类 BSTree 提供了一个成员函数 insert：

```
template<class T>
class BSTree {
public:
    virtual void insert(const T& t);
    //…
};
```

并假设类 BSTree 是这样来实现 insert 函数的：使用递归二分查找树插入算法插入 t 到给

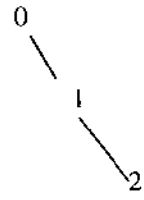
定的 BSTree 对象中。第一，我们不能约束类 BSTree 的派生类中的 insert 函数也使用相同的插入算法；第二，我们也不希望让派生类重新定义 insert 函数来实现完全相同的插入操作。因此，我们可以在文档中如下定义 BSTree::insert 的语义：

使用递归二分查找树的插入算法插入值 t 到给定的树中，派生类可以使用不同的插入算法来重新定义函数 insert；派生类的插入算法在使树保持是一棵二分查找树的前提下，可以任意地重新排列树中的节点。

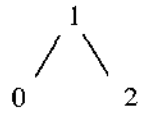
其次，当操作一个指向类 B 的指针或者引用时，程序员往往只能求助于文档化的继承语义；这里假设我们的程序库还包含下面的函数：

```
void insert_several(BSTree<int> & bst) {
    bst.insert(0);
    bst.insert(1);
    bst.insert(2);
}
```

如果调用这个函数时，bst 引用的树是一棵空树，并且这棵树是类型为 BSTree<int>的对象，那么当 insert_several 函数返回的时候，这棵树的外形大体如下图所示：



然而，我们并不能假定这棵树就一定是上面这个图形。如果 insert_several 函数传递的是一个指向 BSTree 的派生类的引用，那么这个派生类就有可能使用另一种不同的插入算法。假如 bst 引用的是一个 AVLTree<int>对象，那么 insert_several 将会产生外形如下的树：



我们最多能假定的内容只能是：当函数 insert_several 返回的时候，bst 引用的树将会是一棵包含值 0、1 和 2 的二分查找树。

再次，由于类 AVLTree 公共派生自类 BSTree，所以类 AVLTree 的实现者必须确保 AVLTree::insert 函数符合 BSTree::insert 函数的继承语义。此外，AVLTree 的实现者还必须定义 AVLTree::insert 的继承语义，以便 AVLTree 的派生类在满足 AVLTree::insert 继承语义的条件下，也就相应能满足 BSTree::insert 的继承语义。例如，AVLTree 的实现者可以用下面的继承语义作为 insert 函数的继承语义：

使用在某些参考书中描叙的AVL树插入算法，插入值t到AVL树中。类AVLTree的派生类可以用其他的插入算法重新定义insert函数；只要在插入结果仍然是一棵AVL树的前提下，派生类的插入算法可以任意地重新排列节点的位置。

3.4 继承的障碍

在这一节里，我们将讨论类继承的障碍。首先，成员函数的存在将妨碍类的继承性，我们将在3.4.7小节讨论这个问题；其次，在这一节里讨论的其余障碍都是针对非接口类的。接下来，我们还可以看到，对于类的继承性设计，接口类要比非接口类简单得多。

3.4.1 非虚成员函数

继承性一个最常见的障碍就是继承的基类含有一个非虚成员函数。譬如，如果BSTree::insert没有被声明为虚函数：

```
template<class T>
class BSTree {
public:
    void insert(const T& t);    //非虚成员函数
    //...
};
```

于是派生类将不能重新定义这个insert函数（也就不能获得所期望的新行为）。为了避免这个问题，我们可以考虑把所有（拟继承的）基类的成员函数都声明为虚函数；然而，这样的做法却会降低程序的效率（我们在4.4.2小节将会看到虚函数的效率低于非虚函数的效率）。因此，我们更愿意只把那些派生类需要重新定义的函数声明为虚函数。

预知基类的哪些函数才是派生类需要重新定义的函数是困难的。假设我们如下实现类BSTree:

```
template<class T>
class BSTree {
private:
    class Node {
    public:
        T t;
        Node* left;
        Node* right;
        Node(const T& _t) : t(_t) { }
        //...
    };
    Node* root;
    //...
};
```

嵌套类Node描述树中的节点；root是指向根节点的指针，或者当树是空树的时候，root

的值将为0。现在假设我们如下实现 insert 函数：

```
template<class T>
class BSTree {
public:
    virtual void insert(const T& t) {
        doinsert(t,root);
    }
    //...
private:
    void doinsert(const T& t, Node*& n);
    //...
};
```

函数 insert 调用了 doinsert 函数，后者实现二分查找树的递归插入算法：

```
template<class T>
void BSTree<T>::doinsert(const T& t, Node*& n) {
    if(n == 0)
        n = new Node(t);
    else {
        if(t < n->t)
            doinsert(t, n->left);
        else
            doinsert(t, n->right);
    }
}
```

注意到大部分的插入工作都是在 doinsert 里完成的。而且，鉴于上面 insert 函数的这种实现方法，对于需要重新定义插入操作的派生类，现在需要重新定义的是 doinsert 函数，而不是 insert 函数。因此，doinsert 函数就应该为虚函数，而 insert 函数为非虚函数也可以。我们在下面把 doinsert 函数从私有（private）类型改变为保护（protected）类型，因为 BSTree 的派生类需要存取这个函数：

```
template<class T>
class BSTree {
public:
    void insert(const T& t);
    //...
protected:
    virtual void doinsert(const T& t, Node*& n);
    //...
};
```

（实际上，把 doinsert 函数声明为私有函数并不是必需的——因为 C++允许子类改写基类的虚私有成员函数；然而，改写基类的私有函数是一个很不好的做法）

另外，每个基类的析构函数都应该声明为虚函数（参考练习 3.5）。因此，我们声明 BSTree 类的析构函数为虚函数：

```
template<class T>
```

```
class BSTree {
public:
    virtual ~BSTree();
    //...
};
```

3.4.2 过度保护

继承的另一个障碍就是对类成员的过度保护。在前一节，我们声明嵌套类 Node 为私有嵌套类，但某些类型的二分查找树可能需要给节点增加额外的字段（Node 的成员变量，也称为域）。例如，表述线性二分查找树的类 Threaded_bstree 可能需要增加一个 thread 字段：

```
template<class T>
class Threaded_bstree : public BSTree<T> {
protected:
    class Node : public BSTree<T>::Node {
                                                    //错误，不能存取-
                                                    //BSTree<T>::Node
    public:
        Node* thread;
        //...
    };
    //...
};
```

因此，我们应该把 BSTree::Node 声明为 protected 嵌套类，而不是 private 嵌套类：

```
template<class T>
class BSTree {
protected:
    class Node {
        //...
    };
    //...
};
```

为了使基类具有很好的继承性，基类的设计者应该充分了解用户需要继承的类究竟是什么类型的。

为了避免过度保护，我们可以考虑把目标基类中所有成员都声明为公共（public）或保护（protected）成员。然而，protected 成员在将来程序库新发布的版本中，如果不破坏代码兼容性，那么将不可以被改变或者被删除，这是 protected 类型的不足之处，（而 private 成员就没有这个缺点）——所以 protected 成员有可能破坏用户代码（我们将在 7.4 节详细讨论代码兼容性）。因此，我们可以只对那些派生类需要存取权的成员声明为 protected 类型。在下面的例子里，我们假设派生类只需要存取 doinsert 成员函数、Node 嵌套类和 root 成员变量：

```
template<class T>
class BSTree {
protected:
```

```
    class Node {
    public:
        virtual ~Node();
        //...
    };
    Node* root;
    Virtual void doinsert(const T& t, Node*& n);
    //...
};
```

在上面代码中，我们还把用户现在可以继承的嵌套类 Node 的析构函数声明为虚函数。

预知类中哪些成员应该成为 protected 成员是很困难的。例如，假设类 BSTree 中还包含一个 ninserts 成员变量，它用来计算所给树中执行插入操作的次数，那么 BSTree 的派生类是否需要存取 ninserts 呢？对这个问题我们将很难判断。

3.4.3 模块化不足

继承的另一个障碍是模块化不足。为了举例说明这个障碍，我们来考察派生类重新定义 BSTree::doinsert 的典型方式。先考虑二分查找树中一个称为 red-black 树的类别[CLR90]，在 red-black 树中，每个节点包含了一个说明该节点究竟是 red 还是 black 的域[1]（成员变量），这个域用来保证这种树保持了以下的平衡特性（在此我们不必关注使用的方式）：从根节点到叶子节点的最长距离最多只是从根节点到叶子节点最短距离的 2 倍。于是，在执行每个插入或者删除操作之后，red-black 树要执行一次重新平衡操作，用来重新得到上面的特性。

下面代码描述用户如何从类 BSTree 派生出表述 red-black 树的类 RBTree：

```
template<class T>
class RBTree : public BSTree<T> {
protected:
    class Node : public BSTree<T>::Node {
    public:
        bool isred;
        Node(const T& t);
    };
    void doinsert(const T& t, BSTree<T>::Node*& n);
    virtual void rebalance(Node* n);
    //...
};
```

函数 rebalance 执行根节点为 n 的树的重新平衡操作。考虑到以后的派生类可能需要重新定义 rebalance 函数，程序员在这里应该把这个函数声明为保护类型的虚函数（protected and virtual）。

下面是类 RBTree 中 doinsert 函数的代码：

[1] red_black 树中的每个节点同时包含了指向其父节点的指针。为了简化我们的讨论，我们抛弃了指向父节点的指针。

```
template<class T>
void RBTree<T>::doinsert(const T& t, BSTree<T>::Node*& n) {
    if(n == 0) {
        Node* m = new Node(t);  //  行1
        n = m;
        rebalance(m);
    }
    else {
        if(t < n->t)
            doinsert(t, n->left);
        else
            doinsert(t, n->right);
    }
}
```

上面的代码和基类的 doinsert 函数代码有两个重要的差别：首先，行 1 代码创建的对象是一个 RBTree::Node 对象，而不是 BSTree::Node 对象；其次，在插入新节点时，调用了 rebalance 函数。

注意到 RBTree::doinsert 函数和 BSTree::doinsert 函数的大部分代码是相同的（语法上和语义上相同）。这就存在着复制代码操作，而复制相同的代码很乏味、容易产生错误并且代码显得臃肿，也会令代码的可维护性变得更加复杂；因为当复制的代码由于某种原因需要进行更改时，维护者就必须相应更改这段代码的所有拷贝。一个具有很好继承性的 BSTree 类将会使派生类的实现者复制尽可能少的代码，甚至不复制代码。如果 BSTree 类的设计者能够意识到：许多二分查找树都需要创建不同类型的节点，并对这些树执行重新平衡操作（rebalance），那么这些设计者可能会有远见地设计如下的 BSTree 类：

```
template<class T>
class BSTree {
protected:
    virtual Node* newnode(const T& t) {
        return new Node(t);
    }
    virtual void rebalance(Node* n) {
        //在 BSTree 类中，这里是空的。
    }
    //...
};

template<class T>
void BSTree::doinsert(const T& t, Node*& n) {
    if(n == 0) {
        n = newnode(t);
        rebalance(n);
    }
    else {
```

```
            if(t < n->t)
                doinsert(t, n->left);
            else
                doinsert(t, n->right);
        }
    }
```

在上面的 BSTree::doinsert 函数中，我们取代了派生类需要调用虚函数改写的代码。而且，和以往一样，派生类可以重新定义这些函数：

```
template<class T>
class RBTree : public BSTree<T> {
protected:
    Node* newnode(const T& t) {
        return new Node(t);
    }
    void rebalance(BSTree<T>::Node* n) {
        //...
    }
    //...
};
```

有了上面的设计，派生类的程序员就不必要再去复制代码了。

从上面代码可以看出，函数 RBTree::newnode 返回一个指向 RBTree::Node 对象的指针，而 BSTree::newnode 返回一个指向 BSTree::node 对象的指针。如果基类有一个虚函数 *f*，并且 *f* 的返回类型是一个指向某个类 *X* 的指针（或引用），于是对于 *X* 的派生类的 *f* 函数，返回一个指向 *X* 派生类的指针（或引用）类型将是合法（类型安全）的。这种改变虚函数返回类型的能力称为函数返回类型协变（function return type covariance），这也是 C++相对较新的特性之一。

在上面的代码中，我们把函数 RBTree::rebalance 的参数从 RBTree::Node 类型改变为 BSTree::Node 类型。然而遗憾的是，这个改变使 RBTree::rebalance 函数在存取 *n* 指向的 RBNode 对象的 isred 域时，就必须执行一个向下转型操作。但相对于可以不用复制代码而带来的好处而言，这个不足就显得微不足道了。

如果希望确保派生类的程序员将永远不需要复制代码，那么我们应该把派生类需要改变的代码段分开出来，并且单独构成一个虚函数。然而，这种做法有两个缺点：第一，把这些代码段分开出来构成独立的虚函数将会耗费程序员大量的时间，而且，在我们还没有掌握关于派生类的足够信息时，要分解出这种虚函数是不可能的；第二，每个附加的虚函数调用都会对基类和派生类的效率产生负面影响。例如，在函数 BSTree::doinsert 中调用空函数 BSTree::rebalance 就是一个虚函数调用，这个调用在原先的 BSTree::doinsert 函数里并不存在。虽然单个虚函数调用对 BSTree 类和它的派生类的效率并没有明显的影响，但如果一个只追求不复制代码的设计有很多的这种虚函数，就会使程序的效率非常地慢。

3.4.4 friend 关键字的使用

过于依赖 friend 关键字将会降低类的继承性。为了说明这个观点，我们下面用友元关系（friendship）来实现类 BSTree。从 3.4.1 小节到 3.4.3 小节，我们把类 Node 嵌套在类 BSTree 里面，而另外一个设计就是让类 Node 成为一个单独的类（在这个过程中，我们把类的名字改为 BSNode），如下代码所示：

```
template<class T> class BSTree;
template<class T>
class BSNode {
protected:
    T t;
    BSNode(const T& t);
    friend class BSTree<T>;
};
template<class T>
class BSTree {
    //现在没有内嵌 Node 类
    //...
};
```

由于友元关系，类 BSNode 的实现属于类 BSTree 的实现细节，而且为了防止非 BSNode 的派生类偶然地存取 BSNode 的成员，所以我们把 BSNode 的成员都声明为 protected 类型，并让 BSTree 类作为它的友元类。

类 RBTree（BSTree 类的派生类，由上一节可知）的实现者同时也会让类 RBNode 继承类 BSNode，因为类 RBTree 需要存取节点基类 BSNode 的成员。然而，友元关系（friendship）是不能继承的——也就是说，虽然 BSTree 是 BSNode 的友元类，但 BSTree 的派生类并没有因继承而成为 BSNode 类的友元类。为了解决这个问题，我们有两个办法：第一个办法可以声明 BSNode 的成员为 public 类型——从而也就牺牲了 protected 类型的优点并使友元关系（friendship）毫无意义（原因见上段），第二个办法就是用户必须在 RBNode 类内部增加对 BSNode 类成员的存取函数：

```
template<class T>
class RBNode : public BSNode<T> {
protected:
    T& tval() {return t;}
    //...
};
```

相比之下，上一节没有使用友元关系(friendship)的设计就不需要用户实现存取函数。

3.4.5 成员变量过多

目标基类成员变量太多也是继承性的一个障碍，幸运的是这个障碍在实际中很少出现。假设基类 BSTree 的节点除了包括指向左子节点和右子节点的指针外，还包括了一个指向父

节点的指针：

```
template<class T>
class BSTree {
protected:
    class Node {
    public:
        Node* parent;
        Node* left;
        Node* right;
        //...
    };
    //...
};
```

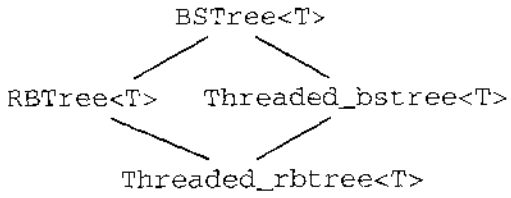
假设某个用户希望实现一棵以空间优化为首要目的二分查找树 Compact_bstree，在类 Compact_bstree 里面，每个节点只存储 2 个指针（指向左子节点和右子节点的两个指针），而不是如上面 BSTree 类的 3 个指针。然而遗憾的是，以空间优化为首要目的的二分查找树却不能从类 BSTree 继承，因为派生类并不能删除从基类继承下来的、不需要的数据成员。

为了解决这个问题，对派生类有可能不需要的成员变量，基类的设计者就应该避免定义这些成员变量。然而遗憾的是，决定那些数据成员集合是派生类所需要的数据成员集合将会是很困难的，而且这种集合往往会是空集。

如果我们把类 BSTree 设计为接口类（见 3.2.1 小节），那么类 Compact_bstree 的实现者就能够从接口类 BSTree 派生出类 Compact_bstree；然而，由于 BSTree 是接口类，类 Compact_bstree 就不会继承任何类 BSTree 的实现，而不得不从头开始实现每个函数。

3.4.6 非虚（Nonvirtual）派生

假设我们决定在程序库中同时提供 BSTree 类和 RBTree 类，并且考虑到用户可能会如下派生出 Threaded_bstree 类和 Threaded_rbtree 类：



可惜的是，虽然 Threaded_bstree 对象是某种类型的 BSTree 对象，并且 Threaded_rbtree 对象是某种类型的 RBTree 对象，但是上面这种设计方法是行不通的。因为 RBTree（见上一节）是从 BSTree 非虚派生的，于是 Threaded_rbtree 对象将会包含两个 BSTree 类子对象，而这明显不是用户想要实现的。所以，用户应该让类 RBTree（和类 Threaded_bstree）虚派生自类 BSTree。

然而，虚派生也有两个缺点：第一，假设类B没有任何虚成员函数，也没有继承任何虚成员函数，那么要从类B向下转型为它的派生类D将是不可能实现的，但向下转型有时候又是不可避免的；第二，在大多数系统上，针对非接口类而言，与非虚派生相比，虚派生在效率上将处于下风。

因此，我们只对那些如果不使用虚派生就会给用户带来问题的类进行虚派生。况且预知哪些类应该使用虚派生也是很困难的。在我们上面这个例子里，如果经过RBTree类从BSTree类进行多重派生（直接派生或者间接派生）是不太可能的，但我们又希望程序具有最大的效率，那么我们就应该让类RBTree从类BSTree非虚派生。

3.4.7 妨碍继承的成员函数

有时，某些成员函数的存在对包含这些函数的类而言是合情合理的，然而，这些成员函数却削弱了类的可继承性。假设我们的程序库提供了一个名为Graph的类（graph，图，是点和连接点的边的集合）：

```
template<class T>
class Graph {
public:
    class Node{ /* ... */ };
    //...
};
```

为了使用户可以直接操作Graph中的节点，我们把Graph中的节点类Node声明为public，而不是BSTree里面的Protected。

我们还需要提供一个函数，它给Graph增加一个新的节点：

```
template<class T>
class Graph {
public:
    virtual void addnode(Node* n);
    //...
};
```

调用addnode函数可以把n指向的节点添加到所给的图中；我们还把addnode函数声明为虚函数，这样就可以增强Graph类的继承性，而Graph类其余的函数将使用户可以在点之间增加连接点的边。假设我们定义addnode函数的继承语义如下：

添加n指向的节点到所给的图形中。

在这之后，假设我们的用户希望实现一个用于描述树的类Tree；显而易见，树也是图的一种，因此用户希望可以让类Tree公共派生自类Graph。然而遗憾的是，这是不可能办到的，因为给树图增加一个没带任何边（包括入边和出边）的节点后，这棵树也就不再成其为树了，也就是说，Tree类不能满足上面addnode函数的继承语义。

另一个可以让类Tree从类Graph继承的方法就是：松动addnode函数的继承语义，改变后的继承语义如下：

添加 n 指向的节点到所给的图中，或者什么事也不做。

在上面允许把 addnode 函数定义为空函数之后，Tree 类就可以满足 addnode 函数的继承语义了。然而，这种解决问题的办法也将导致这样的现象：当用一个指向 Graph 的指针或者引用调用 addnode 函数的时候，这个调用并不能保证一定可以添加节点：

```
Graph<T>* g;
//...
g->addnode(n)          //n 指向对象的节点究竟添加了没有？ 不知道。
```

在上面的代码中，g 可能指向 Graph 的派生类对象，而在派生类中，addnode 函数可能已经重新定义为什么事情也不做。可见，如 addnode 这样的接口将违反直观性并且容易产生错误。

一个让 Tree 类派生自 Graph 类具有更好继承性的办法就是，在 Graph 类中去掉 addnode 函数：

```
template<class T>
class Graph {
    //现在没有 addnode 函数。
    //...
};
template<class T>
class Tree : public Graph<T> {
    //现在没有 addnode 函数。
    //...
};
```

我们当然可以在 Graph 类的派生类中引入 addnode 函数：

```
template<class T>
class Graph_with_addnode : public Graph<T> {
    void addnode(Node* n);
    //...
};
```

我们可以把这种情况推广一下，如果类 X 具有 3 个成员函数 f、g 和 h，而每个函数都会降低 X 的继承性，那么我们就可以如图 3.1 所示改进（细化）Graph 的继承性的方法，来恢复类 X 的继承性。在图 3.1 中，那些只满足函数 f 和 g 继承语义的派生类就可以从类 X_with_f_and_g 派生；而那些只满足函数 f 继承语义的派生类可以从类 X_with_f 派生；以此类推。

上面的设计方法虽然增强了继承性，但这种精细继承体系的方法具有很快就使类体系变得过于复杂的缺点。在下面这个精细继承体系里，类的数目会随着降低继承性函数的数目而指数式递增；为了减少类的数目，设计者应该找到提供足够继承性的最原始的细化。例如，假设我们知道在某些 X 的派生类中，对于 f 函数和 g 函数，只要能满足其中一个的继承语义，就同时能满足另外一个的继承语义，并且没有用户需要同时满足 3 个函数继承语义的派生类，那么我们就可以舍弃图 3.1 的继承体系，而使用如图 3.2 所示的更简单的继承体系。

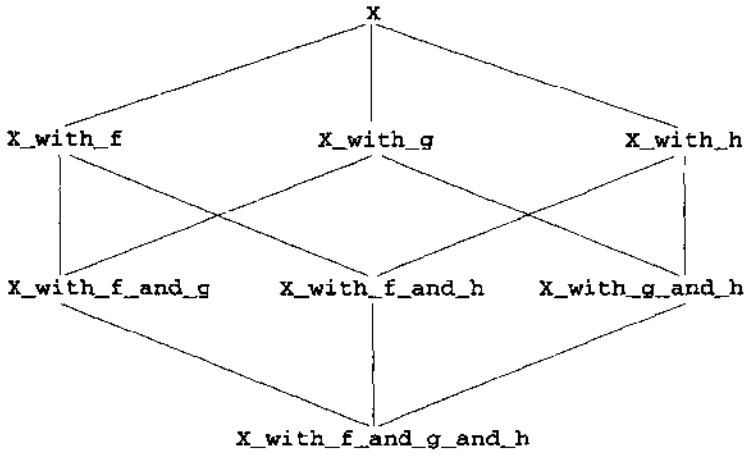


图 3.1　一个精细的继承体系

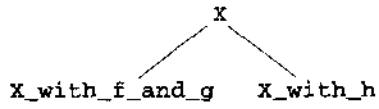


图 3.2　一个比上图稍微粗糙的继承体系

3.5　派生赋值问题

如前所述，抽象类是至少具有一个纯虚函数的类，这个纯虚函数可以是继承而来的，抽象类可以只用作基类。对于一个类，如果不是抽象类，那它就是具体（concrete）类。

从前面的 3.4.3 小节到 3.4.6 小节，我们从具体类 BSTree 派生出具体类 RBTree。而具体类之间的互相派生很容易会引发所谓的派生赋值问题[1]。为了能够充分理解这个问题的实质，假设我们希望类 BSTree 成为 nice 类（见 2.3 节），每个 nice 类都提供了一个基本的赋值运算符：

```
template<class T>
class BSTree {
public:
    const BSTree<T>& operator=(const BSTree<T>& t);
    //...
};
```

相似地，假设类 RBTree 的设计者也希望类 RBTree 成为 nice 类，因此也需要提供一个基本赋值运算符：

[1] 事实上，派生赋值问题是这样一个更一般问题的特例：派生类设计者有时需要参数类型可以协变，而这是 C++所不允许的。然而，当联系到赋值运算符时，这种问题出现得最多。

```
template<class T>
class RBTree : public BSTree<T>{
public:
    const RBTree<T>& operator=(const RBTree<T>& t);
    //...
};
```

现在考虑下面的代码：

```
void buggy_func() {
    RBTree<int> r, s;
    BSTree<int> b;
    BSTree<int>* p = &r;
    *p = b; //行1
    *p = s; //行2
}
```

上面行 1 和行 2 的赋值运算都调用了 BSTree::operator=，而不是 RBTree::operator=。因此，在行 1，BSTree 对象 b 将会把自身的值赋给 r 对象的 BSTree 部分；而在行 2，s 对象值的 BSTree 部分将会赋给 r 对象的 BSTree 部分，但这些明显都不是任何人希望执行的操作。

我们可以通过声明 BSTree 的赋值运算符为虚函数，从而来解决上面这个问题。如果真的这样做了，还必须改变派生类赋值运算符的参数类型，来适应这个操作：

```
template<class T>
class RBTree : public BSTree<T> {
public:
    //令赋值运算符函数接收一个 BSTree 类型参数对象
    const RBTree<T>& operator=(const BSTree<T>& t);
    //...
};
```

遗憾的是，上面这个赋值操作符已经不再是基本的赋值操作符；更加重要的是，定义这个操作的语义也变得不可行了。例如，如果 Threaded_bstree 类不一定是 RBTree 类的子类，那么把一个 Threaded_bstree 对象赋给一个 RBTree 对象的操作意味着什么呢？

为了解决这个问题，我们可以考虑给 RBTree::operator=添加下面的前提条件：参数 t 引用的对象的类型必须是 RBTree 类的后代派生类。那我们是否应该检查这个前提条件呢？如果我们不检查这个条件，那么我们在 buggy_func 函数的赋值操作中的成功之处就只在于：只是把不符合需要的行为转变成没有定义的行为，而没有定义的行为同样是不符合需要的[1]。另一方面，如果检查这个前提条件，我们必须给 RBTree::operator=的实现添加 dynamic_cast（动态转型），但这样的实现是很不好的，而且，它只是把 buggy_func 函数里面赋值操作的错误转变成运行时错误。然而，在我们的设计中，我们更愿意这种赋值操作引发的错误是编译时错误，而不是运行时错误。

对于派生赋值问题，一个更好的解决方法就是不要从具体（concrete，相对于 abstract）

[1] 译注：这里不符合需要的行为是指，前 3 段文中“这些明显都不是任何人希望执行的操作”的内容，就是把子类对象赋给基类对象。没有定义的行为，指把没有前提条件的某些类（非基类子类）赋值给基类。

基类公共派生出具体子类。在我们这个例子里，可以使用接口类来把类的继承体系转变成如图 3.3 所示的继承体系。因为在图 3.3 中，类 RBTree 是私有（private）继承自类 BSTree 的，所以 buggy_func 函数内的赋值运算符就只会出现编译时错误了。然而，这种设计观点也是有缺点的：它要求更多的类和更多的派生类，从而也就使类的理解和使用变得更加困难。在实际应用中，大多数程序库的设计者通常使用比这更加简单的设计，而并不会太过在意派生赋值问题。

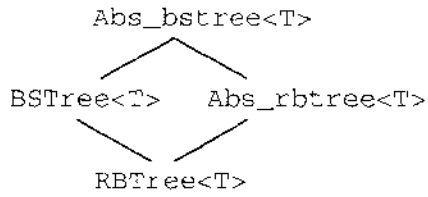


图 3.3　转变后的不存在派生赋值问题的继承体系，类 RBTree 从类 BSTree 私自派生

3.6　允许入侵（用户修改源代码）继承

如果类的用户具有存取类源代码的权限，那么一些继承的障碍将会很容易地得到解决。例如，如果程序库的用户需要在派生类中重新定义基类函数，而基类函数并不是虚函数；那么，具有存取源代码权限的用户就可以在基类函数中增加一个 virtual 限定符，来使基类函数成为虚函数，之后在派生类中改写这个虚函数；或者当需要在派生类中存取基类成员的时候，这些用户也可以把那些在程序库中声明为 private 类型的变量改变为其他的类型（如 protected 或 public）。这种为了支持基类的继承而改写基类源代码的做法被称为入侵继承（invasive inheritance）。当然，具有入侵继承权限的程序员会带来给基类的实现引入错误的危险；而且，入侵继承将使包含可入侵基类的程序库在新发布的版本中，复杂性不断上升。

因为编写可继承的类是很困难的，而当用户具有存取源代码的权限时，某些继承性的障碍就可以很容易被克服，所以你可能会考虑分发所有或者部分程序库的源代码。当然，分发源代码也有它自身的许多缺点（见练习 3.11）。

3.7　总结

提供可扩展性是有代价的。有时候，一个合理的替代办法就是在设计可扩展 C++程序库时，提供所有用户需要的功能，因此用户也就没有必要扩展程序库的类了。

在更普遍的情况下，用户还是需要扩展性的。在 C++中，可扩展性一般是通过继承来提供的。我们应该适当地定义类的继承语义，并且假设在整个程序库中，这些继承语义对编写

可扩展类是必不可少的。于是，成功继承的重担压在了编写派生类的用户身上了——因为如果公共派生类型不符合它基类的继承语义，继承性的实现将是天方夜谭。

对那些没有仔细考虑可继承性的类而言，要从它们派生出子类是很困难的。下面就是可继承性的一些障碍：

- 非虚成员函数；
- 对数据成员和成员函数的过度保护；
- 成员函数的模块化不足；
- friend 关键字的使用；
- 过多的数据成员；
- 非虚派生；
- 阻碍继承的成员函数。

接口类由于并不会出现大多数继承性的障碍，所以对那些继承性具有重要地位的程序库而言，所有的类都应该实现为接口类，用户也就可以很容易地继承这些类的接口。

3.8 练习

3.1 下面哪些选项具有可扩展性？

a. 图书馆书的集合；

b. 圣经中书的集合；

c. 拉丁语的词汇；

d. 因特网中所有电子邮件地址的集合。

3.2 通过改写 3.4.3 小节的 BSTree::newnode 函数，给出 3.2.2 小节的 Map 类的实现。

3.3 如何对私有派生进行改写呢（如 3.2.2 小节的 Map 类从 BSTree 类私有派生）？如何把私有派生改写成公共派生，或者具有内嵌结构（一个对象被内嵌于另一个对象里面）的类体系结构呢？

3.4 如果一个类沿多条路径从接口类 X 派生，并且其中有多于一条的路径是从类 X 非虚派生的，那么将会有什么问题发生？

3.5 解释为什么用户要继承的基类的析构函数要声明为虚函数？

3.6 假设在这个练习和下面的练习（3.7）中使用 3.4.3 小节的类 BSTree 和类 RBTree 版本

a. 实现函数 BSTree::constains，返回一个 bool 值，用来指出函数的参数（const T&类型）是否在树中已经存在？

b. 在实现函数 BSTree::constains 之后，类 BSTree 还是可继承的吗？

c. 实现函数 BSTree::remove，如果给定的值存在的话，就从 BSTree 中把这个值删去。

BSTree::remove 会返回一个 bool 值，用来指出该值是否存在。

d.（*）实现 RBTree::remove 函数，牢记在 red-black 树中实现 remove 操作还必须重新平衡 red-black 树，那么这样的 BSTree 类是可继承的吗？

3.7 在这个练习中，回顾 3.2.2 小节的 Map 类：

a. 用 RBTree 类来实现 Map 类。

b. 在 a 部分的解决方法中，你是否从 RBTree 里派生了一个类？如果已经从类 RBTree 派生了一个类，那么类 RBTree 的继承性如何？

3.8 在这一章里，我们看到了如何使用继承性来使 C++程序库具有可扩展性。而且，继承性也可以使程序库变得更容易使用；假设 ShinyBrite 牙膏制造厂的化学师希望能够在一个牙膏模拟程序中，操纵 Toolpaste 对象。譬如，他们希望能够通过指定牙膏的化学成分来构造 Toolpaste 类。

a. 解释为何下面的 Toothpaste 类版本的创建是很难使用而且容易产生错误的（其中类 Quantity 描述牙膏化学成分的数量）：

```
class Toothpaste {
public:
    Toothpaste(
        Quantity fluoride,
        Quantity oil_of_peppermint,
        Quantity saccharin,
        Quantity red_dye7,
        Quantity blue_dye10,
        /* 某些品牌的牙膏具有更多的参数 */ );
    //...
};
```

b. 为了给复杂的基类提供一个简单的接口，某些程序员会使用 feature filtering（特性过滤）表达式，而且这种过滤是与派生类的使用是息息相关的。假设 ShinyBrite 儿童牙膏的每种不同的产品都包含着相同数量的氟化物（fluoride）、糖精(saccharin)和 10 号蓝色素(blue_dye10)，那么派生自 Toothpaste、具有特性过滤的 Childrens_toothpaste 类如何使程序库能够更容易使用呢（我们将在 8.8 节进一步讨论这个问题）？

3.9 考虑在 3.4.5 小节结束时讨论的类 Compact_bstree，给出如何设计类 BSTree 和它的继承体系方法，从而令 Compact_bstree 类的实现能够尽可能多地重用 BSTree 类的实现。

3.10 假设你有一个 Polygon 类，你希望从 Polygon（多边形）类中可以派生出一个 Rectangle(长方形)类。

a. 假设 Polygon 类有一个成员函数 add_vertex，这个函数给多边形添加一个顶点；又由于类 Rectangle 是继承自类 Polygon 的，这个函数会导致什么问题发生呢？

b. 给出一个继承体系，这个继承体系可以让类 Rectangle 从 Polygon（或者某些你设计来代替 Polygon 的类）成功派生出来。

3.11 对一个程序库而言，分发源代码的缺点是什么？

3.9 参考文献和相关资料

Halbert 和 O'Brien[HO87]详细地阐述了继承的使用方法。Martin[Mar91]、Nackman 与 Barton[NB94]和 Linton 与 Pan[LP94]讨论了 C++程序库中接口类的应用，也讨论了相应的效率问题。

虽然他们都没有使用模块化不足（undermodularization）这个术语，但 Kiczales 与 Lamping[KL92]和 Weide、Ogden 与 Sitaraman[WOS94]都给出了这个现象很好的例子。派生赋值问题首次由 Meyers[Mey94a, Mey94b]提出。精细继承体系问题在 Johnson 与 Rees[JR92]里有详细的讨论。

Schwarz[Sch90]讨论了 C++作为面向对象语言的缺点，其中包括派生赋值问题。

多态的概念和阻碍继承性的成员函数问题是比较有吸引力的话题，除了我们在这一章里讨论的内容，还有很多关于它们的内容；感兴趣的读者可以阅读 Cline 和 Lomow 的书[CL95]里的第 10 章和第 11 章的内容，也可以参考 Baclawski 和 Indurkhya 的关于这个话题简短而精彩的讨论，后者（Indurkhya）引用的作品也相当不错。

Carroll[Car92b]第一次提出了 invasive inheritance（入侵继承）这个概念，并给出了关于这个现象的很好例子。

第4章 效率

与其他单一的原因（包括肯目恩蠢的想法）相比，效率（指无法有效地获得效率）是计算机犯罪的主要原因。

——Wm.A.Wulf

效率是可重用代码的本质特性。在这一章里，我们将讨论如何设计和实现可重用代码，来提高下面资源的使用效率：程序创建时间（包括编译时间、链接时间和实例化时间）、代码大小、运行时间和内存占用量。

4.1 效率和重用性

大多数程序员认为效率是重用性的一个本质特性。对于任何程序库，无论设计得如何优雅，如果严重地降低了程序的效率，那么它还是不能得到重用的。就如程序库的使用，即使只是增加了少量的编译时间，但对于要编译几千个源文件的开发团队，如果这些文件的创建需要几天时间的话，他们还是不能忍受这微小的额外时间的。因此，对于编写实时应用程序的程序员，他们拒绝使用导致应用程序不能满足时间约束的程序库；对于嵌入式系统（例如银行自动柜员机）的程序员，他们也不会使用可执行文件占用的空间比目标机器的可存储字节数还要大的程序库。

在理想情况下，开发者希望可重用代码可以和自己编写的代码一样：具有一样快的速度和一样的代码大小。但遗憾的是，这只是一种理想情况，并且几乎是不能实现的。我们都清楚，可重用代码必须是在很多情况下都能得到很好应用（见 1.3.2 小节）的代码；而在某个特定情况下最大化地优化可重用代码，必定导致在其他情况下这些代码失去很好的重用性。幸运的是，一段可重用代码的成功与否，并不在于它是否有和自己编的代码一样高的效率。现

在让我们来考虑运行时间，一个大家经常凭经验引用的法则就是：程序90%的运行时间花费在10%的代码上面（或许你听说过80%运行时间花费在20%的代码之中）：于是，如果一段可重用代码并没有用在这10%或20%运行频率很高的代码之中，那么这段可重用代码就不需要最优化了。而且，90/10或者80/20法则对于其他的效率量度问题（如创建时间、内存占用量等）同样是适用的。此外，只要可重用代码不对效率产生过大的影响，程序员肯定会认识到使用可重用代码所带来的其他好处的（如他们不需要再次自己编写相同的代码；可重用代码是已经经过很严格测试的代码，因此也就具有更好的可靠性，并提供了很好的文档资料等等）。

总而言之，可重用的C++代码应该尽量少耗用下列资源：程序创建时间、代码大小、运行时间、空闲存储空间（free store，也叫空闲存储区、自由存储空间）和堆栈空间；我们将在下面各节分别讨论这些资源。

4.2 程序创建时间

对于给定的源文件，在它上面运行宏预处理程序后产生的代码称为翻译单元。创建一个C++程序的时间包括两部分：编译所有翻译单元所需要的时间和把这些翻译单元链接成一个可执行程序所需要的时间；其中实例化程序所用模板的时间包含在上面的编译时间和链接时间里面。在这一节里，我们给出了一些程序库实现者用于减少程序创建时间的技术。

4.2.1 编译时间

对于使用程序库中某些资源（如函数）的程序，减少它的编译时间的一个最简单的技术就是：最小化#include程序里面的代码。例如，假设我们的程序库给用户的窗口系统（window system）提供了一个接口，而且这个接口定义在程序库头文件Window_system.h里面：

```
Windows_system.h :
    class Widget {
        //...
    };
    class Button : public Widget {
        //...
    };
    //程序库头文件中还有许多其他的类...
```

和大多数窗口系统的头文件一样，假设上面这个头文件也包含很多的代码（直接的代码和间接的代码），并且我们希望在自己程序库的头文件里面使用上面的Button类：

```
Ourlib.h:
    #include <Window_system.h>
    class Our_window {
    private:
        Button button1;
        Button button2;
```

```
    //...
};
```

遗憾的是，如果以这种方式来实现我们的程序库，那么所有包含 Ourlib.h 头文件的翻译单元——甚至那些没有使用窗口系统的翻译单元——都将包含 Window_system.h 头文件，这将大大影响编译速度。解决的方法是：只声明类的名称（如 Button），而不包含头文件，从而来提高编译速度：

```
Ourlib.h:
    Class Button;        //取代 #include<Window_system.h>
    Class Our_window {
    Private:
        Button* button1;
        Button* button2;
        //...
    };
```

我们还必须相应地将 button1 和 button2 的类型从 Button 类型转变成 Button*类型，于是现在对 Button 对象的所有存取操作都会通过一个间接层来进行，即要先寻找 Button 类的定义。因此，这种技术只是在编译时间和运行时间之间寻求平衡（这里减少了编译时间，增加了运行时间）。

有时候，我们可能会把这种技术推广到声明类以外的一些事物（如函数）。例如，假设 Window_system.h 包含以下 create_buttons 函数的声明，这个函数返回一个指针，指向新创建的包含 n 个 Button 对象的数组：

```
Button* create_buttons(size_t n);
```

如果在 Ourlib.h 里调用 create_buttons 函数，我们就可以试着只声明 create_buttons 函数，而不包含 Window_system.h:

```
Ourlib.h:
    //取代#include<Window_system.h>
    class Button;
    Button* create_buttons(size_t n);
    Class Our_window {
    Public:
        Our_window() {
            button1 = create_button(1);
            //...
        }
        //...
    private:
        Button* button1;
        //...
    };
```

上面的代码看起来非常地简单，但它可能包含了一个错误：传递给 create_buttons 的数字 1 是 int 类型，而不是函数声明中的 size_t 类型。如果 Window_system.h 还包含下面的声明：

```
Button* create_buttons(int n);
```

那么我们对 create_buttons 函数的调用就能解决上面的错误。如果 Window_system.h 现在并没有重载 create_buttons 函数，它也可以在以后的版本中重载这个函数，我们的代码也就只能在新版本中顺利通过编译。但由于以后重载的函数可能会改变代码实体，所以这个函数实现的功能或许将不再是我们原先希望的功能。为了避免这个问题，程序员应该只直接声明类的名称——而不能是函数——来取代包含头文件。

4.2.2 实例化时间

随着模板被广泛地使用，C++程序将在实例化模板上面花费更多的创建时间；而且，对于某些 C++用户而言，如何减少模板实例化时间已经成为一个亟待解决的问题。开发团队也非常注重创建时间，因为他们总是编译程序、运行测试程序、调试程序、改变程序的某些部分并再次编译等等。因此，对用户而言，最小化模板实例化时间已经成为编写可重用代码的一个很重要的目标了。

1．简单的技术

下面两种简单的技术可以用来减少实例化时间：程序库预实例化模板（或者预实例化某些模板）和定义函数模板为内联（inline）。其中，在库文件中预实例化模板是减少程序库用户实例化时间最明显的方法。例如，假设我们提供了 3.2.1 小节的二分查找树模板：

```
template<class T>
class BSTree{
    //...
};
```

如果在程序库的实现中我们使用了 BSTree<String>，或者发现大多数用户都创建了 BSTree<String>，那么我们就可以预实例化 BSTree<String>，并把实例化后的结果文件放入库档案文件里面。这种预实例化模板机制是高度依赖于操作系统的，我们将在 9.7 节讨论这个问题。

某些模板必须在程序库内部预实例化。假设我们在程序库内部使用了 BSTree<Private>，其中 Private 是一个只在这个程序库实现中使用的类型，并且在任何程序库的公共头文件中都不提供这个类型。于是，如果用户创建了一个使用这个程序库的应用程序，那么实例化器将找不到 Private 类型的实现，由于 Private 类型上面的性质，用户根本就不能实例化 BSTree<Private>，而必须由程序库自己对它进行预实例化。

在大多数系统中，另一个减少用户实例化时间的技术就是定义函数模板为内联的：

```
template<class T>
class BSTree {
    int _size;
public:
    int size() const {return _size;}
    //...
};
```

在上面代码中，size 函数返回树中节点的数目，其中节点的数目存储在成员变量_size 中。size 函数由于被声明为内联函数，所以编译期间在每次调用的地方，它的实例化速度将非常地快，这也是大多数系统实例化函数的最快方式。当然，这个技术只适用于那些可以声明为内联的函数。

通常，对于任何编程技术，只要能够减少实例化代码的大小，它就可以减少实例化时间。下面的两个技术是 hoisting（即提升，指在原来类体系的基础上，声明一个非模板的基类）和使用指针容器（pointer containers）。

2. Hoisting（提升）

假设由于参数的原因，BSTree::size 函数不能被定义为内联函数：

```
template<class T>
class BSTree {
public:
    int size() const;
    //...
};
template<class T>
int BSTree<T>::size() const {
    return _size;
}
```

然而，上面 size 函数的实现并不依赖于 BSTree 的模板参数。理论上讲，C++编译系统应该能够发现：size 函数的所有特殊化目标代码（即由于模板参数不同引起的不同代码）都是一样的，从而编译系统只实例化了 size 函数的一个特例；但现今的编译系统几乎都不能够执行这种优化。于是，我们可以通过 hoisting（提升）size 函数的定义到一个新创建的非模板基类里，来高效地执行这种优化：

```
class BSTreebase {  //用于提升的非模板类
public:
    int size() const;
protected:
    int _size;
};
int BSTreebase::size() const {
    return _size;
}
template<class T>
class BSTree : public BSTreebase {
    //现在是继承自基类的 size 函数
    //...
};
```

我们还必须同时提升成员变量_size；而且，我们把_size 声明为 protected 类型，是因为模板类 BSTree 的派生类可能需要存取这个变量。

在某些时候，对于模板类的成员函数，即使在它依赖于模板参数的条件下，我们也可以

提升这个函数。假设我们用下面独立的类 Node 来描述 BSTree 树中的节点：

```
template<class T>
class Node {
    T t;
    Node* preorder_successor;
    //...
};
```

我们希望提供一个函数 BSTree::thread，用于前序[1]线性化所给定的树——特别地，当函数 BSTree::thread 返回的时候，树中每个节点的 preorder_successor 域指向它的前序后继者，下面是 BSTree::thread 函数的伪代码：

```
template<class T>
void BSTree::thread() {
    Node<T>* prev = 0;
    Node<T>* n = 0;
    for(树中前序的每个节点) {
        if(prev)
            prev->preorder_successor = n;
        prev = n;
    }
    if(n)
        n->preorder_successor = 0;
}
```

thread 函数的实际实现可能会使用树的前序遍历递归算法，而不是上面的 for 循环。而且从上面的代码可以看出：thread 函数的代码实体依赖于模板参数 T（在 pre 成员和 n 成员的声明语句中）。然而，从概念上讲，对树进行线性化并不依赖于树中节点存储值的类型；因此，只要再做一些工作，我们就应该可以提升 thread 函数。首先，让我们从一个非模板基类派生出模板类 Node:

```
class Nodebase {          //为了提升而设计的非模板类
protected:
    Nodebase* preorder_successor;
};
template<class T>
class Node : public Nodebase {
    T t;
    //...
};
```

我们同时也提升了 preorder_successor 成员变量。现在我们就可以根据 Nodebase 类重写 thread 函数，来提升 thread 函数：

```
class BSTreebase {
public:
    void thread();
```

[1] 译注：这是前序的意思等同于前序遍历中的前序。即先访问根节点，再访问左子树，最后访问右子树。

```
    //...
};
void BSTreebase::thread() {
    Nodebase* prev = 0;
    Nodebase* n = 0;
    //剩下的代码和原先在 BSTree::thread 中的代码一样
    //...
}
```

3. 指针容器

减少实例化代码数量的另一个技术就是提供指针容器（pointer container）。假设我们的程序库提供了一个 List 模板：

```
template<class T>
class List {
public:
    T head() const;
    List<T> tail() const;
    Void insert(const T& t);
    //...
};
```

List<T>是一个值的类型为 T 的链表；函数 head 在 List 不为空的前提下，返回 List 的第一个值；函数 tail 同样在 List 不为空的前提下，返回一个原先删除第一个值后的 List；函数 insert 将在 List 的表头插入值 t。

例如，假设用户创建了一个 List<int>、一个 List<String>、一个 List<Widget*>和一个 List<Blidget*>（其中 Widget 和 Blidget 是用户自己定义的类），并且对上面每个 List 都调用了 head、tail 和 insert 函数；那么，每个函数都将被实例化 4 次，总共对函数实例化了 12 次。在用户的程序里，一个被广泛使用的类（如 List）可能会用不同的类型来进行实例化，同时也就必须实例化许多函数，最后导致生成许多（List）成员函数的代码。

除了普通的 List 模板之外，如果提供了适用于指针的 List 版本，我们就可以减少用户需要实例化的代码数量：

```
template<class T>
class Plist {
public:
    T* head() const;
    Plist<T> tail() const;
    Void insert(T* t);
    //...
};
```

Plist<T>描述一个值的类型为 T*的链表。而且从上面代码可以看出：Plist::head 的返回类型为 T*类型，而不是 T 类型；相似地，insert 函数接收一个 T*类型的参数；用户现在将创建一个 Plist<Widget>对象，而不是以前的 List<Widget*>对象。由下面的代码可知，Plist 的实现使用了一个 List<void*>对象 rep，并且 Plist 的每个成员函数调用 rep 的相应的 List

函数：

```
Template<class T>
Class Plist {
Public:
    T* head() const {return (T*)rep.head(); }
    Void insert(T* t) {rep.insert(t); }
Private:
    List<void*> rep;
    //...
};
```

如果可以成功地把 T*类型的值转化为 void*类型的值，并且可以转化回来，从而得到链表节点的原始值，那么上面的代码就可以实现我们希望的功能。虽然有些 T 类型并没有上面的转化特性（例如，指向成员的指针类型），但普通用户所希望插入容器的类型一般都具有上述转化特性（关于 Plist 究竟只适用于哪些类型，应该提供清楚的文档资料）。

为了实现 tail 函数，我们使用了 Plist 的私有构造函数：

```
template<class T>
class Plist {
private:
    Plist(const List<void*>& r) : rep(r) { }
    //...
};
template<class T>
Plist<T> Plist<T>::tail() const {
    Return Plist<T>(rep.tail());
}
```

从上面可以看出，Plist 成员函数的代码实体是很简单的；实际上，这些函数都是内联函数的候选函数，而且内联（我们在 4.2.2 小节进行讨论）通常是实例化函数最快的方式；另外，我们还可以预实例化 List<void*>对象。因此，比起那些使用 List<Widget*>和 List<Blidget*>的程序员，使用 Plist<Widget>和 Plist<Blidget>的程序员只需要实例化更少的代码，而且 Plist 的用户将看到程序创建时间也相应减少了。

指针容器虽然减少了实例化时间，但却有一个显著的缺点：容器类和指针容器，虽然从概念上讲，在 C++的类型系统中应该是关联的，但实际上 C++类型系统并没有实现这种关联。假设 List 和 Plist 的用户编写了下列代码：

```
template<class T>
void insert_twice(const T& t, List<T>& l) {
    l.insert(t);
    l.insert(t);
}
```

虽然对于任何类型的 List，我们都可以调用这个函数，但这个函数却不能使用 Plist 类型为参数。为了提供和 Plist 类型相同的功能，我们需要另一个函数：

```
template<class T>
```

```
void insert_twice(const T& t, Plist<T>& l) {
    l.insert(t);
    l.insert(t);
}
```

某些程序库设计者曾经尝试过解决这个问题，他们通过继承把一个类和这个类的指针容器相关在一起：

```
//这种使用继承的设计并不是一种好的设计
template<class T>
class Plist : public List<void*> {
public:
    T* head() const {return (T*)List<void*>::head();}
    //...
};
```

然而遗憾的是，这个设计并不是类型安全的。如考虑下面用户的代码，其中X和Y是不相关的类型：

```
void insert_y(List<void*>& l) {
    l.insert(new Y);
}
int main() {
    Plist<X> l;
    insert_y(l);          //在List l中插入一个类型为Y*的值
    X* x = l.head;  //出问题了，我们得到一个X*类型的值
    //...
}
```

上面的代码既没有编译时错误，又没有创建时错误；然而，它把一个Y*类型的值转化为一个X*类型的值，而这并不是一个安全的转型。

4.3 代码大小

程序的代码大小是指由该程序所有翻译单元组成的所有目标文件中的字节数。如果一个程序具有很大数量的代码，那么它的可执行文件可能需要比较多的用于存储的磁盘空间和用于执行代码的内存空间。（我们说可能，是因为一个程序究竟需要多少空间还与很多依赖于系统的因素有关，例如程序是否使用了动态链接库，可执行文件中实际的字节数和文件在磁盘和内存中是如何存储的等等）

如果某个C++程序库的使用将导致用户程序产生大量的代码，那么对那些没有足够磁盘空间和内存的程序员而言，他们将不能使用这个程序库。因此，为了提高代码的重用性，C++程序库的设计者应该尽量减少程序库代码的大小。

4.3.1 源文件分割

一个很出名的减少代码大小的技术就是：对于两个很大的函数，如果某些程序只需要其

中的一个函数，而不需要另外一个函数，那么我们就应该把这两个函数分别放于不同的程序库实现文件里面。我们把这种技术称为源文件分割技术（智能编译系统没有使用这种技术的必要，因为它们本身就可以实现相似的技术，但现今几乎不存在这样的编译系统）。现在请考虑4.2.2小节的BSTreebase类：

```
class BSTreebase {
public:
    int size() const;
    void thread();
    //...
};
```

假设由于参数的原因，上面两个函数都很大（实际上，size函数通常都是很小的）。而且可能存在某些程序，它们只需要其中的一个成员函数，而并不需要另外一个成员函数，那么源文件分割技术将指导我们把这两个函数的定义分别放在不同的文件里面。另外，如果thread函数需要调用私有成员函数dothread，并且只有thread调用dothread函数，那么源文件分割技术也要求我们把dothread函数和thread函数放在同一个文件里面。

当我们考虑模板的时候，事情将变得更加复杂一点。假设4.2.2小节的BSTree模板类并不是从类BSTreebase继承而来的：

```
template<class T>
class BSTree {
public:
    int size() const;
    void thread();
    //...
};
```

当我们编写BSTree类的模板成员函数的定义时，为了使用户的模板实例化程序可以存取这些函数，我们就必须把这些函数的定义放在公共头文件里面，而且，用户的模板实例化程序还需要进一步组织模板源代码（我们将在9.7节详细叙述用户的这种需求），因此，在这种情况下，我们就不能在这些函数定义的目标代码中进行源文件分割，因为我们根本就没有这个选择的权利。

如果对一个大的程序库强行应用源文件分割，那么可能会产生问题。首先，现有的许多用于创建和操纵目标代码的档案文件的程序不能处理包含几百个小文件的档案文件。为了解决这个问题，现有的许多C++程序库都提供几个比较小的档案文件，而不是只提供一个大的档案文件。然而，这个解决方案的使用要慎重一些，因为它要求（程序库）用户切记按照正确的顺利链接所有的档案文件。

其次，如果我们在程序库中创建了调试变量（见5.3.1小节），那么许多库的目标文件将会包含许多重复的信息。如果程序员接着使用我们程序库的调试变量来创建某个程序的调试变量，那么程序的代码数量将会急剧增大，而这有背源文件切割的初衷。

为了解决上面这两个问题，我们可以放宽对源文件切割的要求：把那些可能会在同一个

程序里使用的函数集合都放在同一个文件里面。但在这种情况下，不再能够保证仍然可以得到最小化的正确链接时间开销。

4.3.2　外联的（outlined）inline

那些希望减少用户代码数量的程序库设计者应该谨慎地杜绝外联的 inline（指内联失败导致的结果）。记得 inline 关键字只是一个用于 C++编译器的标志。当前所有的 C++编辑器对其能够内联的函数都是有限制的。例如，某个编译器可能不能内联（inline）任何包含 goto 语句的函数，或者任何超过 15 行语句的函数；某些函数调用也是不能内联的——例如，不能内联具有递归调用的函数。我们将在下面的 4.4.2 小节看到：大多数编译器甚至不能内联最简单的虚函数，因为当通过某个对象调用这个虚函数的时候，我们很难静态地决定这个对象的实际类型。

如果函数 f 被声明为内联函数，但未能在翻译单元每个调用的地方都对 f 进行内联扩展，那么在这些翻译单元中，许多编译器将会通过内部链接创建一份外联 f 函数的拷贝。如果在 n 个翻译单元中都创建了这样的外联拷贝，那么可执行文件将会包含 n 份 f 函数的拷贝。因此，如果程序员没有对内联函数的声明持小心谨慎的态度，那么代码的数量将会由于外联的 inline 而变得异常庞大。

让人觉得奇怪的是：某些具有优化功能的编译器，都很难断定某个重要函数是否需要内联。只有那些相当好的编译器，才能充分考虑函数被调用位置的上下文——而不是只局限于函数的复杂度——来判定所给函数调用是否需要内联。

程序库设计者可以使用的一个安全的方法就是：如果存在程序库用户编译器会在典型的调用位置外联 f 函数的可能性，那么我们就不应该把 f 函数声明为内联函数。而且，一个好的 C++程序库的附加文档也会随同给出：在哪些情况下，编译器不会内联函数。

4.3.3　模板特化大小

在大多数情况下，C++程序库减少由库本身产生的代码的最好的方式就是：在用户程序中，减少库模板特化所需要的代码。而 4.2.2 节讨论的提升技术和使用指针容器技术也都是减少程序创建时间，并很好减少用户代码数量的好方法。

程序库应该尽可能少地使用模板。考虑 4.2.2 小节的模板类 BSTree 的实现，与其使用 BSTree<char>、BSTree<long>和 BSTree<int>，我们可以只在程序库中使用 BSTree<long>，相应也节约了代码空间；但这样做有一个缺点：类 BSTree 的实现将会变得比较难以理解。

相似地，假设在程序库的开发过程中，它的实现需要一个 List 模板、一个 Sequence 模板和一个 Linked_list 模板；所有的模板都表述相互之间只具有少许区别的链表，并在程序库中使用相同的参数进行实例化，比如说 List<String>、Sequence<String>和 Linked_list<String>。于是，我们可以使用这其中的一个模板来代替所有的模板，或者可以创建一个具有上面 3 个模板各自优点的新模板。

4.4 运行时间

大多数程序员认为运行时间是衡量效率优劣最重要的尺度，这是很有道理的。对于一个太大、或者创建时间太长、或者消耗太多内存空间的程序，我们将不得不忍受等待的焦熬，直到这个个相当慢的过程被结束（或者中止）之后才能继续运行。

减少运行时间最有效的方式就是尽可能地使用高效的算法。例如，假设要实现一个对大型链表排序的函数，那么我们就应该尽可能地使用几个有名的quicksort算法（[Baa88,CLR90]）中的一个，或者使用Wegener[Weg93]中描述的heapsort算法。

有许多方法可以用来衡量算法的运行时效率。我们可以考虑算法的最差效率、平均效率或者分段具体效率以及其他的效率。我们还可以观察I/O操作的次数，浮点数计算的次数，或者任何其他衡量效率的操作。

一旦我们确定实现任务的最佳算法——依照我们所选择的衡量尺度——我们就必须尽量高效地实现这个算法。而关于如何高效实现算法的完整技术讨论已经超出本书的范围（可以参见[Ben82]），在这里我们只讨论几个对C++代码而言相当重要的技术。

4.4.1 内联（inlining）

通过对适当选择的函数进行内联而节省运行时间是相当重要的。然而，选择正确的内联函数往往要比人们预想中的困难许多。

1. 内联的神话

程序员们往往都赞同这样的看法：内联以增加代码数量为代价，加快了程序的执行速度，减少了程序的运行时间。大多数情况下内联的效果确实如此，但并不是所有的情况都是这样。假设某个函数的代码数量非常地小，那么内联扩展所占用的空间将比建立一个函数调用和一个函数返回所需要的空间少；就是说，在这种情况下，程序中使用内联扩展的地方越多，相应地使用程序行调用的地方越少，程序中所包含的代码就会越少。这就不符合上面以增加代码数量为代价的前提。

而且，内联也可以降低程序的执行速度。假设一个大函数、几个小函数或者中等大小的函数在一个循环里面实现内联，从而内联将导致循环中代码实体急剧增加，而且超过了指令缓存的大小（假设在一台有指令缓存的机器上），那么这个程序的执行速度比起没有实现内联的程序的执行速度还要慢。另外，同时各种机器的其他特性对内联的效果也有不利影响。

程序员还可能会这样考虑：只有小的或者中等大小的函数才能是内联的候选函数。但事实是任何函数——无论函数大小——都可以是内联的候选函数。如果在某个特定的位置内联某个函数，那么就可以省去那些由于没有内联而花在参数调入和调出上的时间。现在假设在程序执行频率较高的代码模块中（并且指令缓存并不是目标机器的瓶颈），程序只在一个地方调用某个较大的函数，那么内联这个函数就不会增加代码的数量，但可能会增加程序的运行时间。

2．内联的优缺点

由于使用内联而节省的运行时间将不仅仅包括由于不使用内联而花在压入和弹出参数（也可能是拷贝参数）的时间、返回参数值的时间，从函数调用和返回的时间。而且，内联还经常给程序优化器创造了机会。对于某个程序，内联可能会创建更大的基础块（关于基础块的定义和这里使用的另外一些最优化的概念，请参阅[ASU86]）。如果在函数调用时使用了常量参数，那么内联将有利于常量在这整个程序中的传送。而且，在内联函数的实体内，我们将可以省去在调用上下文中进行计算的某些子表达式。从而，在包含内联扩展的函数中，寄存器也可以高效地分配存储空间。

然而，内联有两个潜在的缺点：首先，对于那些由于内联扩展产生的代码多于不使用内联而因函数调用和返回所需要的代码的函数，如果我们在程序中多处使用这个函数，那么程序的代码数量将会增加。代码增加的数量取决于函数调用的次数和函数本身的大小，而且函数本身的大小不能只取决于函数的定义。例如，考虑下面代码：

```
void f() {
    g();
    h();
};
```

函数 f 看起来很小，但是如果 g 函数或者 h 函数是内联函数的话（或者两者都是内联函数），f 的实际代码大小就可能会很大。另一个例子，考虑下面代码：

```
class D : public B {
public:
    inline D() { }
    //...
};
```

上面代码中，D 的构造函数是空函数，但它包含了一个对 B 的默认构造函数的隐式调用。如果 B 的构造函数是内联函数的话，那么 D 的构造函数至少会和 B 的构造函数一样大。另外，某些编译器还会添加一些其他的开销给 D 的构造函数，如调用 malloc 函数分配存储空间等。

内联另一个潜在的缺点是：对内联函数实现的更改往往并不可以保持链接兼容性——这就是说，这样的更改需要用户重新编译他们的代码。（我们将在7.5节进一步讨论链接兼容性。）

3．使用内联的时机

很明显，如果我们需要保持某个函数的向前链接兼容性（而不是要获得更快的程序执行速度），我们就不应该把这个函数定义为内联函数。如 4.3.2 小节所讨论的内容，许多编译器拒绝内联某些函数，或者在某种上下文情况下，不会内联某些函数。因此，如果在多个翻译单元中出现许多外联(out-of-line)拷贝的话(指内联由于上面原因失败后执行的操作)，那么相对于内联带来的好处，内联带来的坏处将会更多。而且，大多数编译器不能内联对虚函数的调用，因为它很难静态地决定调用内联虚函数的对象的实际类型（我们将在 4.4.2 小节进一步讨论这个话题），因此把虚函数声明为内联函数会使代码数量高度膨胀。于是，对于那些链接

兼容性并不是程序的主要问题，而且大多数编译器可以对它们实现内联的函数，我们是否应该进行内联呢？这主要取决于内联这个函数所带来的下面两方面的影响：代码大小和运行时间（表 4.1）。

表 4.1　　是否应该内联某个函数

内联对运行时间的影响	内联对代码大小的影响		
	减少	影响不大	增加
减少	是	是	可能
影响不大	是	可能	不
增加	可能	不	不

虽然因未能在具体用户的系统上测试某些应用程序，而不能总是知道内联对这些应用程序的代码大小和程序执行速度两方面有何影响，但是我们往往可以知道关于一般用法和用户运行程序的系统方面的信息，从而利用这些信息来做出正确的决定。

- 如果内联 f 函数既可以减少用户程序的代码大小，又可以减少程序的执行时间，那么我们当然应该内联这个函数。相似的，如果内联 f 函数可以减少用户的代码或者程序的执行时间，并对另外一个不产生重大的影响，我们也应该内联这个函数。
- 如果内联函数 f 既增加用户的代码大小，又增加程序的执行时间，或者增加其中之一，而对另一个并没有产生重大的影响，那么我们就不应该内联这个函数。
- 如果内联函数 f 可以减少程序运行时间，但增加用户程序的代码大小，或者减少用户程序的代码大小，但增加程序运行时间，那么是否对函数 f 使用内联要取决于在这两个影响中，每个影响的程度和我们的用户更注重哪一部分资源。
- 如果内联对代码大小和运行时间的影响都是可以忽略不计的，那么是否内联函数 f 要取决于内联是否有利于程序的进一步优化。

4.4.2　虚函数

虚函数有两部分的开销：首先，与调用一个外联的非虚函数相比，调用一个外联的虚函数需要增加一些额外的机器指令。除非这个函数调用的频率非常高，否则这个效率的影响并不大。其次，也是更重要的影响因素，在大多数调用上下文中，现今的编译器并不能内联虚函数。为了说明这个问题，考虑 4.2.2 小节开头给出的 BSTree::size 函数，我们现在把这个函数如下声明为虚函数：

```
template<class T>
class BSTree {
    virtual inline int size() const {
        return _size;
    }
```

```
    //...
};
```

如果上下文可以很容易地静态确定是在 BSTree 对象调用这个函数 BSTree::Size（而不是在其派生类的某个对象上调用），那么大多数编译器就可以内联 BSTree::size 函数：

```
BSTree<int> b;
//...
int sz = b.size();        //这里大多数编译器都可以内联。
```

另一方面，如果不能（或者很困难）决定 BSTree::size 是否在 BSTree 对象本身上调用，那么现有的大多数编译器就不能内联 BSTree::size：

```
void f( const BSTree<int>& b) {
    int sz = b.size();  //大多数编译器都不能内联：这里的 b 是一个引用类型，b 面
                          的不是
    //...
}
```

因此，基于现在的编译器技术，当我们需要把一个函数声明为虚函数时，我们就放弃了在大多数上下文内联这个函数的能力。关于是否内联一个函数或者把这个函数声明为虚函数问题，我们在表 4.2 给出了总结。

- 如果内联成员函数 f 是至关重要的，而用户又需要改写 f，那就需要寻找某种重新设计并可以解决这个冲突的办法。幸运的是，在实际中这种冲突出现的几率很小。
- 如果内联成员函数 f 是至关重要的，并且用户不需要改写 f，那么就应该把 f 声明为非虚内联函数。
- 如果内联不太重要，并且用户需要改写这个函数，那么就应该把 f 声明为非内联的虚函数。
- 最后，如果内联和改写都不是很重要，那么 f 就应该被声明为非虚函数，才能避免调用虚函数所花费的开销。

表 4.2　　成员函数是否应该声明为内联函数或者函数

用户是否需要改写	内联是否至关重要	
	是	否
是	重新设计程序库	非内联的虚函数
否	非虚内联函数	非虚函数，见表 4.1

当然，要想知道哪些函数是用户需要改写的函数是非常困难的（见 3.4.1 小节）。当有这方面疑问（即是否需要改写）的时候，我们就应该把函数声明为非虚函数；否则当真正需要改写函数时，用户肯定会不知所措（除非函数的源代码已经分发，用户可以得到源代码，见 3.6 节）。

几乎没有函数可以既声明为虚函数，又声明为内联函数。因为当这类函数的调用不能被

内联的时候，将有可能在每个翻译单元中都生成一份这类函数的外联拷贝（见 4.3.2 小节和 4.4.2 小节）。

4.4.3 返回引用

一个广受提倡的用于减少运行时间的技术就是返回引用，而不是返回值，因为返回值将需要高昂的拷贝开销。考虑 4.2.2 节的 List::head 函数，所有 head 函数的调用者——包括那些只是检查表头节点值的用户——都要承受把节点值拷贝出去的开销：

```
List<Widget> l;
//...
cout << "first Widget is " << l.head();      //拷贝了一个 Widget.
```

为了避免这样使用的拷贝开销，我们可以修改 head 函数，让它返回一个指向内部存储表头值的拷贝的引用：

```
template<class T>
class List {
public:
    const T& head() const;        //返回一个引用
    //...
};
```

如果 List 类如上面一样定义，那么用户代码将不会拷贝一个 Widget 对象。我们还在声明 head 函数时，让它返回一个 const 引用，因为返回一个非 const 引用将允许用户在不使用 cast 转型去除 const 关键字的情况下就可以改变 const List 中存储的值：

```
void whoops(const List<Widget>& l) {
    Widget newvalue;
    //...
    l.head() = newvalue;          //uh-oh! 如果 head 函数返回一个非 const
                                  //对象，这条语句是合法的，但不是一个好的
                                  //主意。
    //...
}
```

对于任何返回引用的程序库函数，程序库文档都应该给出关于引用存活期（指引用的有效时间）的说明。例如，我们如下给出 head 函数的文档：

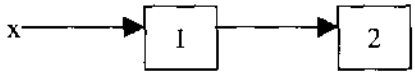
head 函数返回的引用将保持有效，直到引用指向的值从所给的 List 中删除。

返回引用有两个缺点。首先，它使用户的代码更加容易产生错误。回想一下 4.2.2 小节的 insert 函数和 tail 函数：insert 函数把一个节点插入到 List 的表头，tail 函数返回除了头节点之外的表尾，现在让我们考虑下面代码：

```
List<int> l;
l.insert(0);
const int& i = l.head();
l.insert(1);          //i 仍然是有效的。
l = l.tail();         //i 仍然是有效的。
l = l.tail();         //i 现在变成无效了。
```

当 i 变成无效之后，任何试图使用引用 i 的操作都会是未定义的操作。但在返回一个 T 类型（即返回值），而不是返回 T&类型（即返回引用）的 head 函数里，就不会出现上面这种未定义的操作。

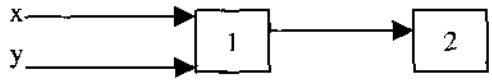
返回引用的第二个缺点是限制了我们实现所给类的方式。例如，我们发现，如果以共享的方式实现类 List，那么大多数用户的代码运行速度将会更快。特别地，让我们考虑 List<int> x，它的底层实现如下图所示：



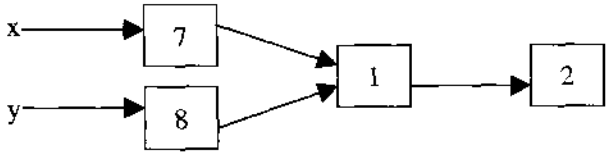
如果用户编写下面代码：

```
List<int> y = x;
```

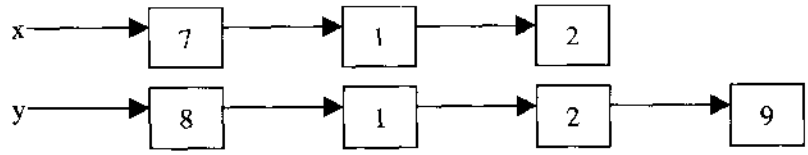
那么 y 将和 x 共享同一个节点：



如果用户在 x 中插入一个 7，并在 y 中插入一个 8，那么底层实现将会如下所示：



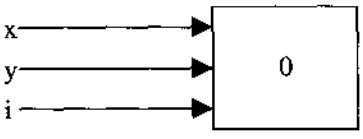
最后，如果用户要在 y 指向的链表尾部追加值 9，那么 x 和 y 将不再共享同一个表尾；因此，我们把所有共享节点都拷贝出来生成一个新的链表，并把 9 附加到生成的新链表的尾部：



现在假设函数 remove_all 将会移除链表中所有的值，函数 append 将会在链表末尾追加一个给定的值。考虑下面的代码：

```
List<int> x;
x.insert(0);
List<int> y = x;
const int& i = y.head();
```

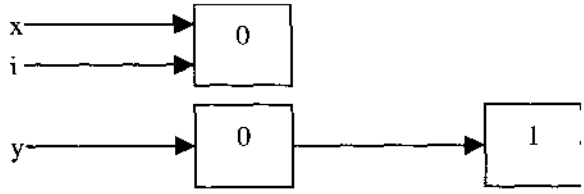
下面是我们上面代码得到的底层实现：



如果我们把 1 追加到 y 的末尾：

```
y.append(1);          //i 仍然是有效的。
```

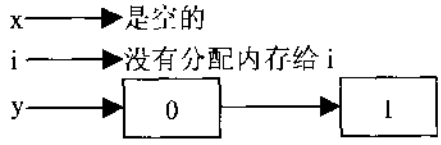
那么 x 和 y 将不再具有相同的表尾，所以 y 的 List 将是一个拷贝的 List。（我们也可以直接在 y 原来的 List 追加一个节点，即让 y 保持原来的链表，而让 x 指向一个新的拷贝。但这个方法并不能解决问题，而只是把问题转移了——假设有一个指向 x.head()函数的引用，那样问题将会出现。）现在的底层实现如下：



假设我们现在改变 x：

```
x.remove_all();       //i 是否依然有效呢?
```

因为程序中并没有其他和 x 共享上面节点的 List 对象，所以 remove_all 操作将会删除 x 中所有的节点。从图形来看，我们将得到以下结果：



在调用了 remove_all 函数之后，引用 i 可能会是无效的。因此，基于上面 List 类的实现，我们根本不能保证（除非我们特别聪明），head 函数返回的引用将会保持有效，直到此引用指向的值从所给的 List（即我们调用 head 函数的 List）中删除。

基于上面 List 的实现，我们对 head 函数唯一可以做的保证只能如下：

head 函数返回的引用将保持有效，直到对任何 List<T>执行了非 const 操作。

遗憾的是，有了上面这个保证，用户程序却会有意无意地操纵无效的引用。所以说，为了加强上面这个保证，来防止用户程序操纵无效的引用，我们往往需要改变 List 的实现，而这种改变通常都会降低 List 的效率。

返回引用可以提高效率，但它可能会使用户代码更加容易产生错误，并且限制类的实现方式。如果函数确实需要返回引用，那么程序库就必须给出关于每个引用存活期的文档。所以说，使用引用的过程必须是非常小心谨慎的，来确保每个引用都是有效的、正确的。

4.5 空闲存储空间（free-store）和堆栈空间（stack space）

从概念上讲，一个可执行的 C++程序的内存分为下面 3 部分：

- 静态存储空间（static storage）存放程序的文本和静态数据。

- 在调用 operator new 函数、malloc 函数和其他一些系统特有的函数时，从内存中分配的存储空间叫做空闲存储空间；而且，空闲存储空间的大小是会随着程序执行的过程不断发生变化的。
- 最后，堆栈空间是指存储自动变量和函数参数的内存；栈存储空间的大小也是会随着程序执行的过程不断发生变化的。

在 4.3 节我们讨论了减少程序代码大小的多种方法，其中也包括减少程序静态存储空间的方法。在下面这一节里，我们将着重考虑减少空闲存储空间和堆栈空间的方法。

4.5.1　使用高效的算法

最小化使用空闲存储空间最有效的方法就是使用高效利用空间的算法。假设我们正在实现一个程序库，打算供 C++编程开发环境使用；我们还希望提供另一个函数，用来编译给定的 C++代码：

```
bool compile(istream& i, const List<String>& incldirs,
    ostream& o);
```

函数读取 i 中的文本，并且尝试编译这段文本；使用 incldirs 作为搜索要#include 文件的路径列表。如果在这个过程没有发生错误，compile 函数会把编译结果的目标代码写入 o 中，并返回 true；否则，compile 函数将返回 false。

我们可以分两个阶段来实现 compile 函数：首先，预处理代码，并把生成的宏扩展文本储存在空闲存储空间中，然后编译这些文本。遗憾的是，上面这个算法使用了大约 s 个字节的空闲存储空间，其中 s 是宏扩展文本的大小；因为对于实际程序，s 很容易达到 500K 甚至更大，所以如果我们的用户并没有足够的空闲存储空间来容纳 compile 函数的实现，那么我们就不得不使用另外一个不同的实现。

与其把宏扩展文本都写入内存里面，还不如把它写入一个文件里面。假设我们实现了下面两个函数：

```
bool preprocess(istream& i, const List<String>& incldirs,
        ostream& o);
bool compile_preprocessed(istream& i, ostream& o);
```

第一个函数先预处理 i 中的文本，如果没有出现错误，preprocess 函数将把宏扩展的结果写入 o 中，并返回 true 值；否则返回 false 值。第二个函数编译 i 中的已经预处理好（假设已经通过第一个函数的处理）的代码；如果不出现错误，compile_preprocessed 将把编译后产生的目标代码写入 o 中，并返回 true 值，否则返回 false 值。

有了上面两个函数，我们就可以如下实现 compile 函数：

```
#include<fstream.h>
bool compile(istream& i, const List<String>& incldirs,
        ostream& o) {
    ofstream otmp("tmpfile");
    if(preprocess(i,incldirs,otmp)) {
        otmp.flush();
```

```
        ifstream itmp("temfile");
        return compile_preprocessed(itmp, o);
    }
    return false;
}
```

compile 函数的上面这个实现使用了文件系统空间---而不是空闲存储空间——来储存宏扩展文本。文件系统空间在大多数系统都是非常充足的。然而，这个实现要求整个宏扩展文本先写入一个文件，然后再从文件中读取出来；而文件 I/O 显然要比内存存取慢很多。

为了使 compile 函数在空间使用和运行速度上都达到较高的效率，我们必须每次在内存中预处理一给定大小的文本块。最后，假设我们设计了下面一个完成这种功能的类：

```
class Preprocessor {
public:
    Preprocessor(istream& i, const List<String>& inclDirs);
    Bool nextChunk(String& s);
    //...
};
```

上面的构造函数在字符流 i 的文本中创建了一个 Preprocessor 对象。只要预处理操作没有结束，nextchunk 函数就会把 s 赋值为下一块的宏扩展文本，并返回 true 值；当流中已经没有文本的时候，它就会返回 false 值。而且，每一块宏扩展文本的大小都不大于相应给定的常数（即预先已经规定的内存块大小常数）。

我们接下来改变 compile_preprocessod 的接口，让它接收一个 Preprocessor 对象，而不是 istream&对象：

```
bool compile_preprocessed(Preprocessor& p, ostream& o);
```

对于 compile_precessed 函数上面的实现，当它需要更多文本的时候，将调用 Preprocessor::nextchunk 函数。现在我们就可以如下实现 compile 函数：

```
bool compile(istream& i, const List<String>& inclDirs,
        ostream& o) {
    Preprocessor p(i, inclDirs);
    Return compile_preprocessed(p, o);
}
```

compile 函数的上面这个实现将会使用一个固定大小的空闲存储空间来储存翻译单元，而并不在乎输入的文本实际上有多大。另外，由于这个实现避免了使用文件系统，它的运行速度也很快。唯一不足的地方就是，这样的实现比起 compile 函数其他的实现要复杂得多。

4.5.2 尽可能快地释放空闲资源

C++程序库通常都应该尽可能快地释放它们占用的资源。考虑上面的 compile_preprocesed 函数，当代码编译的时候，在每个作用域都会有符号表的入口，而每个作用域的符号表都包含着这个作用域内已声明名字的信息。很显然，符号表占用的空间应该分配在空闲存储区域，

那么，程序库最早什么时候可以删除每个符号表呢？这个答案取决于作用域的类型。函数作用域的符号表可以在函数代码编译完之后立刻被删除，因为位于函数定义后面的程序文本的含义并不依赖于函数内部声明的名字。如下所示：

```
void f() {
int i;
//...
    }
    //这里的代码的含义不受 i 和 f 是否存在的影响。
```

另一方面，类作用域的符号表只有在 compile_preprocessed 函数编译完所有要编译的代码之后，才能被删除：

```
class T {
public:
    void f();
};
int main() {
    T t;
    t.f();        //使用了类 T 作用域内声明的函数名字 f。
}
```

要想准确决定资源最快可以被释放的安全时机是很困难的。对于释放资源，通常的解决方式是使用垃圾收集器，但 ANSI/ISO C++并没有要求 C++编译器提供一个垃圾收集器（虽然某些 C++编译器确实提供了垃圾收集器，但并不是所有的 C++编译器都提供；因此，依赖于垃圾收集器的存在将会限制代码的移植性）。

4.5.3 静态对象

如果在程序库的实现中定义了任何静态对象，那么实现者就需要小心处理这些对象所使用的空闲存储空间。考虑 4.5.1 小节讨论的 preprocess 函数，假设由于某种原因，preprocess 函数使用了一个静态变量，用来保存输入文本的当前行：

```
bool preprocess(istream& i, const List<String>& incldirs,
        ostream& o) {
    static String curline;
    while(more text in i){
        curline = nex line of text in i;
        //...
    }
    //...
    return true;
}
```

如上面的伪代码所示，preprocess 每次都读取一行文本，并把它赋给 curline 变量。而且，由于 String 类的实现，这行文本会被储存在空闲存储空间里面。于是，当 preprocess 函数返回的时候，问题就出现了：curline 这时还存储着文本的最后一行；也就是说，用于存储 curline 的空闲存储空间将会被白白浪费，直到下次调用 preprocess 函数或者这个程序执行结束。另

外，对于输入行的长度并没有给予一定的限制，从而对 preprocess 函数浪费的空闲存储空间的大小也就没有限制了。

为了解决这个问题，可以在从程序库函数返回之前，释放静态对象使用的空闲存储空间：

```
bool preprocess(istream& i, const List<String>& inc1dirs,
        ostream& o) {
    //...
    curline = ""     //,释放 curline 使用的空间。
    return true;
}
```

把空字符串赋值给 curline 变量几乎释放了 curline 使用的所有空闲存储空间，只有遗留微不足道的存储空间没有释放。很自然地，通过上面这种方法，对象所使用空间的释放方式取决于对象的类型。

4.5.4 庞大的对象

现在让我们考虑堆栈空间。对于许多系统，一个程序可利用的堆栈空间要比可利用的空闲存储空间少很多。（而可利用的空闲存储空间要比可利用的文件系统空间少很多。）

在任何给定的时刻内，程序使用的堆栈空间总大于栈中所有对象的大小总和（当我们所说的对象的大小，是指对象执行 sizeof 函数后返回的大小）。如果一个程序用完了所有的堆栈空间，往往都是由于下面两个原因中的其中一个造成的：一个错误的存在，导致了无数次的递归；或者在栈中创建一个或多个很大的对象。关于庞大的对象，考虑下面的代码：

```
class Huge {
    char buf[1024 * 1024];      //1 兆字节的对象。
    //...
};
```

如果程序在栈中创建一个 Huge 对象，很可能它（这个程序）会用完所有的堆栈空间。幸运的是，我们很少会创建一个如此大的对象。如果真地想要创建一个这样的对象，或者用户程序可能需要在栈中创建一个这样的对象，那么我们就应该从空闲存储空间分配所需要的内存空间，而不是从堆栈空间分配这些空间：

```
class Huge_in_freestore {
private:
    char* buf;
public:
    Huge_in_freestore() : buf(new char[1024 * 1024]) {
        //...
    }
    ~Huge_in_freestroe() {
        delete[] buf;
    }
    //...
};
```

如 4.5.2 小节所讨论的那样，我们必须牢记要尽快地删除从空闲存储空间分配的对象。在这里，我们在类的析构函数中删除了 Huge_in_freestore::buf 对象。

4.6　效率的权衡

效率和其他每个 C++程序库所希望的特性往往都会互相制约。特别地，用各种可能的方法设计一个高效的程序库通常都会使程序库的实现变得更加困难，或者使程序库的使用变得更加困难。

假设我们是描述图的 Graph 类的设计者；我们希望 Graph 类能够尽可能地高效。无论给 Graph 类选择什么样的接口，我们都需要描述节点和边的类：

```
template<class T>
class Node {
private:
    T t;
    //...
};
template<class T>
class Edge {
public:
    Edge(Node<T>* m, Node<T>* n);
    //...
};
```

在上面的代码中，T 是储存在每个节点中的值的类型。Edge 类的构造函数接收两个指向要连接成这条边的节点的指针（在实际的程序库中，我们可能把 Node 类和 Edge 类嵌在类 Graph 里面）。

假设在每个 Graph 类的节点中，我们需要储存连接到此节点的边的集合；并且假设我们可以使用下面这个通用的 Set 类：

```
template<class T>
class Set {
public:
    void insert(const T& t);
    bool is_empty() const;
    //...
};
```

函数 insert 把 t 插入到集合中；函数 is_empty 当集合为空的时候返回 true 值。我们可以使用 Set 类来描述 Node（节点）的入射边集合：

```
template<class T>
class Node {
private:
    Set<Edge<T>*> incedges;
    //...
```

```
}
```

为了使用户可以遍历他们创建的图，我们需要提供一个函数，它可以返回各个节点入射边的集合：

```
template<class T>
class Node {
public:
    Set<Edge<T>*> incident_edges() const {
        return incedges;
    }
    //...
};
```

遗憾的是，上面这个优雅的设计并不够高效。其中 Set 是一个通用类，从而就很难在许多特定的环境下达到最大化的效率。假设我们发现 Set 类对于实现 Node（节点）的入射边的集合并不够高效；因此，为了令 Graph 类具有更高的效率，我们可以设计一个特定用途的类 Incedges，用来描述节点的入射边集合：

```
template<class T>
class Incedges {
public:
    void insert(Edge<T>* e);
    bool contains(Edge<T>* e) const;
    int size() const;
    //...
};
```

在上面的代码中，函数 inser 会往集合中加入一条边；当给定边（即参数）属于所在集合的话，函数 contain 就返回 true 值；函数 size 返回集合中边的数量。

我们可以如下使用类 Incedges：

```
template<class T>
class Node {
    Incedges<T> incedges;
    //...
};
```

如果我们正确完成了上面所有的工作，那么使用 Incedges 来代替 Set 将会给我们的程序库带来更好的实现效率。遗憾的是，这种更高效率的程序库设计将在下面两方面制约着程序库：使程序库的实现和使用更加困难。我们将在下面的小节里讨论这个问题。

4.6.1 实现更加困难

效率通常都会制约程序库实现的容易程度。譬如，更高效的 Graph 类的设计要求我们：设计、实现和测试 Incedges 类——而这些都是额外的工作，因为如果我们使用 Set 类，我们就不需要做这些工作。而且，使用 Incedges 类取代 Set 类来实现 Node 类，将要求程序库其余的接口实现都发生相应的修改：

```
template<class T>
```

```
class Node {
public:
    bool is_adjacent(Node<T>* n) const;
    //...
};
```

如果节点 n 和给定的节点（*this，即调用 is_adjacent 函数的节点对象）是连接的（就是指由一条边连接起来），那么 is_adjacent 函数将返回 true。

在用 Set 类实现的 Node 类版本里，函数 is_adjacent 的实现是很容易的：如果 operator* 是 Set 类的交集运算符，我们就可以写出下面这个只有一行代码实体的函数：

```
template<class T>
bool Node<T>::is_adjacent(Node<T>* n) const {
    return !(incedges * n->incedges).is_empty();
}
```

上面的代码测试两个集合的交集是否为空集。

假设 Incedges 类并没有提供集合的交集运算符；那么对于使用 Incedges 类的 Node 类版本，实现 is_adjacent 函数将会更加困难。下面是首次编写的代码，部分内容使用了伪代码：

```
template<class T>
bool Node<T>::is_adjacent(Node<T>* n) const {
    for(Edgebase* e in incedges)
        if(e connect n and this node)
            return true;
    return false;
}
```

在上面这个实现里，我们必须迭代整个入射边集合，检查节点 n 是否在某些连接边上面（在实际的代码中，Incedges 类的迭代应该使用某些迭代器类来实现）。而且，虽然上面的伪代码是正确的，但它可能并不是最高效的实现。我们还可以使用下面更高效的方法：与其只迭代（循环）this 节点所连接的边列表，还不如迭代 this 节点和 n 节点中所连接边数较少的列表：

```
template<class T>
bool Node<T>::is_adjacent(Node<T>* n) const {
    Node<T>* sparser_node = this;
    if(n->incedges.size() < incedges.size())
        sparser_node = n;
    for(Edgebase* e in sparser_node->incedges)
        if(e connect n and this node)
            return true;
    return false;
}
```

显然，与使用 Set 类实现的只有一行代码实体的 is_adjacent 函数相比，这个版本的 is_adjacent 函数的实现和使用都要困难得多。

4.6.2 使用更加困难

效率通常会制约程序使用的容易性。假设我们的程序库用户需要一个函数，它返回所给节点的入射边集合；那么，最优雅的接口将会返回一个包含 Edge 对象的 Set 集合：

```
template<class T>
class Node {
public:
    Set<Edge<T>*> incident_edges() const;
    //...
};
```

如果 Node 类是用 Incedges 类实现的，那么 incident_edges 函数的实现就必须进行一次转型：

```
template<class T>
Set<Edge<T>*> Node<T>::incident_edges() const {
    //把 Incedges 对象转型为 Set 对象。
    //...
};
```

而当使用 Set 类来实现 Node 类时，我们并不需要这样的转型。

另一种实现方法是，我们可以把类 Incedges 转移到程序库的接口中，即让 Node::incident_edge 函数返回一个 Incedges 对象，而不是一个 Set 对象；这样我们就可以不进行转型，从而减少了由转型带来的运行时间开销：

```
template<class T>
class Node {
public:
    Incedges<T> incident_edges() const {
        return incedges;
    }
private:
    Incedeges<T> incedges;
    //...
};
```

遗憾的是，我们的程序库现在变得更难使用了。用户现在必须知道具有特殊用途的类 Incedges；而且，用户代码的编写也变得更加困难；就如前面小节的 is_adjacent 函数一样，incident_edges 函数的编写也变得更加困难了。

4.7 总结

对于可重用性代码，效率是一个至关重要的特性。

对开发团队而言，创建时间显得非常重要。下面几种途径都有助于减少程序的创建时间：最小化程序库所包含代码的数量，预实例化模板，定义模板函数为内联函数，提升模板代码和使用指针容器。

程序库实现者可以通过下面两种途径来减少用户的代码数量：分割程序库源文件；在确定该内联不会导致外联现象的时候，把函数声明为内联函数。另外，程序库实现者本身应该尽可能少地使用模板。

对许多用户来说，检验效率最重要的尺度就是运行时间。运行时间通常可以通过适当的内联而得到很好的改善。然而，决定哪些函数可以作为内联函数的过程往往都不是很明确的。另外，返回引用也是一种改善运行时间的技术，但它会使用户代码更加容易产生错误并限制了类的实现方式。

空闲存储空间和堆栈空间都必须高效地加以使用。小心谨慎地使用高效的算法和尽快地释放资源是减少使用空间的最好方法。另外，太大的对象往往都应该在空闲存储空间内进行创建，而不是在堆栈空间进行创建。

遗憾的是，效率和每个 C++程序库所期望的特性几乎都会互相制约。特别地，定义具有最大效率的程序库往往都会使程序库的实现和使用更加困难。

4.8 练习

4.1　如果我们在开发一个程序库，它包含了 4.2.2 小节的 BSTree 类，那么我们是否需要提供一个 BSTree 类的指针容器版本呢？(讨论主要着重于：在实际的开发中，用户是否会经常使用指针类型来实例化 BSTree 类。)

4.2　假设程序库的设计者把程序库的内联函数分成两组：哪些可能使用户代码运行得更快，并且也不会严重增加用户代码的内联函数；哪些可能会使用户代码运行更快，但是会严重增加用户代码的内联函数。设计一个方案，让用户在程序编译时可以在不内联、只内联其中一种、两种都内联之间做出选择。

4.3　假设函数 f 的代码大小是 s，并且在程序中有 c 处调用了函数 f。进一步假设，对 f 执行一次外联（out-of-line）调用所需要的代码大小是 o；假定当 f 被内联的时候，一般没有出现 f 的外联拷贝；而当 f 实在不能被内联的时候，程序将会产生一个 f 的外联拷贝。

a. 如果所有对 f 的调用都是内联调用的话，那么关于函数 f 的总共代码大小是多少？

b. 如果所有对 f 的调用都不是内联调用的话，那么关于函数 f 的总共代码大小是多少？

c. 当所有对 f 的调用从内联调用转化为外联调用的时候，总共的代码大小改变量是多少？

d. 在什么情况下（以术语 c、s 和 o 表示），当所有对 f 的调用都是内联调用时，程序的代码大小可以减少？

4.4　对于 3.2.2 小节设计的 Map 类，假设我们提供了下列操作：

```
template<class X, class Y>
class Map {
public:
    Y& operator[](const X& x);
```

```
    //...
};
```

调用 operator[]将返回一个指向 x 所映射的、内部存储值为 Y 类型的拷贝的引用。operator[]的前提条件是：x 的值必须在 Map 的定义域之内。

a. 假设用户创建了一个 Map<String, Map<String, String> >类型的嵌套映射 m。写出用于给用户为 m 添加<“three”,<“blind”, “mice” > >映射对的代码。

b. 如 4.4.3 小节解释的那样，返回引用也有某些缺点。解释如何使用两个函数来取代 operator[]，这两个函数都不返回引用，这就使用户可以安全地检验和设置 X 类型映射的 Y 值。

c. 基于 b 部分定义的接口，写出实现 a 部分功能（即在 Map 中增加 Map<String, <String, String> >类型的嵌套映射 m）的代码。另外，上面 Map 类接口的两种实现，哪种实现比较容易呢？

4.5 假设我们如下重载了 4.4.3 小节的 List::head 函数：

```
template<class T>
class List {
public:
    const T& head() const;
    T& head();
    //...
};
```

上面这个设计好还是不好？为什么？

4.6 C++程序库应该高效使用的另一个资源是一个程序可以同时打开的文件的最大数目。

a. 假设某个系统允许程序一次打开 64 个文件。那么为何一个只需要同时打开 5 个文件的程序库函数却不能打开所有它所需要数目的文件呢？请给出解释。

b. 是否可以编写一个具有可移植性的函数，它返回当前程序打开文件的个数？

c. 假设程序库函数需要从文件 a 中读取数据，然后从文件 b 中读取数据，再从文件 a 中读取数据。假设我们在读数据前就已经把文件 a 和文件 b 都打开了，并且在读完数据后把它们关闭；于是，我们就同时打开了两个文件。另一种策略是：我们从不同时打开多于一个的文件，即一次只打开一个文件。给出第二种策略的描述，并说出这种策略的缺点。

4.7 如 4.6 节的第二个 Node 类所示，我们让节点的入射边集合对象 incedges 作为 Node 类的内部成员。假设 Graph 的用户希望给定的 Node 可以在多个图中描述，那么给定 Node 节点在不同的图中将连接不同的边。因此，在每个节点中只存储单个入射边集合将是不够的，也不能达到我们所期望的功能。

a. 我们可以同时使用 3.2.2 小节的 Map 类和 4.6 节的 Set 类、Edge 类和 Node 类，然后把这些类放在一起应用，来描述任意的节点、边和图的集合，请给出这种实现方法。（记住用户希望可以在 Graph 类中执行通常的图操作，如图的遍历、查找连接的子图和深度优先查找等等。）

b. 给出一个可能的例子：这个例子通过重用 Set 类和 Map 类，从而使得 Graph 类的使用要容易一些。

c.（*）如果在 Graph 类、Node 类和 Edge 类中重用 Set 类和 Map 类，那么这种重用会对你的程序的效率产生怎样的影响呢？请考虑编译时间、实例化时间、代码大小、内存占用量和运行时间几个方面。

d. 你的程序是否有局部化开销——这就是说，对于那些不需要让 Node 对象在多个 Graph 中存在的用户程序，是否可以和那些不支持 Node 在多个 Graph 中存在的用户程序具有同样的效率？

e. 如果要编写一个可重用的 Graph 类，你会重用 Set 和 Map 吗？

4.9 参考文献和相关资料

Cohen 和他的同事在[CHS91]分析了本章讨论的一些话题。

有许多关于高效算法的好书——如 Aho、Hopcroft 与 Ullman 的[AHU83]、Baase 的[Bas88]、Cormen、Leiserson 与 Rivest 的[CLR90]；另外，Bentley 在[Ben82]中还讨论了关于调整 Pascal 代码的艺术、调整 C++代码所需要的技术等内容。

Koenig[Koe93a]讨论了从函数返回引用的缺点，并给出了之所以为缺点的原因。

Meyers[Mey93a, Mey93b]和 Sprowl[Spr93]讨论了高效使用空闲存储空间的技术。Stroustrup 在[Str94a]的 10.7 节讨论了在 C++执行环境中提供垃圾收集器的优点和缺点。

4.6 节的运行实例的创作灵感是在和 Jonathan Shopiro 的讨论之后才激发的。

第5章

错误

人如果可以在适当的时候脸红，那么脸红的奥秘将会无穷。
——Oscar Wilde

在程序库代码的执行期间，错误往往是不可避免的。为了使程序库能够重用，程序库必须适当地处理各种错误。在这一章里，我们将讨论如何检测错误，以及在错误出现的时候，如何处理错误。错误有许多种不同的种类，因此也就不存在一种可以适用于所有错误的检测和处理策略。其中一类特殊的错误是资源限制（resource-limit）错误；当某些资源用完的时候，就会出现这类错误；而且，产生这类错误通常既不是程序库的错，也不是用户的错。我们还将讨论程序库对资源限制错误的几种响应方式。最后，我们给出异常安全性（exception safty）这个重要的概念——为了确保可重用代码在抛出异常的时候还能够具有正确的行为，我们还需要做些具体的工作。

5.1 可重用代码中的错误

程序库设计者必须处理3种类型的错误。程序库错误（Library error）是指程序库实现错误。当然，程序库的提供者在程序库发布之前，肯定想尽可能多地检测和纠正错误；但是，任何比较大的程序库在发布的时候，都是肯定会包含错误的。

用户错误（User error）是指如何使用程序库方面的错误。例如，假设我们的程序库提供了一个Stack类：

```
template<class T>
class Stack {
public:
    void push(const T& t);
```

```
    T pop();
    int size() const;
    //...
};
```

如果我们要求调用 pop 函数的 Stack 对象是非空的，那么对一个空的 Stack 对象调用 pop 函数，就会产生一个用户错误。

系统错误（system error）是指当用户程序和该程序所运行的系统之间发生交互时所引发的错误。这类错误的发生，部分是系统的责任，部分是用户的责任，其余的可能谁的责任都不是。例如，假设在我们程序库实现的某个地方，我们想要在空闲存储空间创建一个 Stack 对象：

```
Stack<int>* s = new Stack<int>;
```

然而，如果空闲存储空间没有足够的剩余空间，那么这个操作将会失败。

因此，在程序的执行期间，对于每个可能出现的错误，程序库设计者都应该决定如何检测这个错误，并且在错误出现的时候，如何处理这个错误。我们将在 5.2 节和 5.3 节讨论这些问题。

5.2 错误检测

因为错误的种类是多种多样的，所以并不存在一种可以检测出所有错误的策略。然而，还是存在一种可以检测到许多错误的策略：不变性检测。不变性在这里是指某个值永远为真的特性，范围要比平常所指的范围大很多。例如，假设我们要通过下标存取一个数组：

```
char* p = "some nifty string";
int i;
//...
char c = p[i];
```

大致说来，上面代码的不变性是指：在数组的下标存取操作之前，i 的值必须是非零的，而且必须小于 p 指向的数组的长度：

```
char* p = "some nifty string";
int i;
//...
if(i < 0 || i >= strlen(p)) {
    //不变性不满足条件，失败!
    //...
}
char c = p[i];
```

这个错误究竟是程序库错误、用户错误还是系统错误，要取决于程序剩下的代码。顺便提及一下，为了检测错误而引入的测试代码增加了代码的运行时间：时间复杂度从原来的常数级上升为现在的与数组长度线性相关的级别；也就是说，错误检测通常和运行时间互相制约。

实际上，检测下面两种类型的不变性通常就可以检测出很多错误：函数前提条件和表示

不变性。

5.2.1　函数前提条件

函数前提条件也是一种不变性。当函数被调用的时候，它的值将一直为 true　我们已经给出一个函数前提条件的例子，即 5.1 节的 Stack::pop 函数——我们的前提条件是调用 pop 函数的 Stack 对象应该是非空的。另一个有代表性的例子就是下面的 Data 类：

```
class Date {
public:
    Date(const char* datestring);
    //...
};
```

Date 对象描述一个特定年中的特定日期；它的构造函数以一个字符串为参数，创建一个 Date 对象：

```
Date birthday("April 27, 1970");
Date berlin_wall("9 Nov 1989");
```

然而，并不是每个以 null 结束的字符串都可以被解释为有效的日期：

```
Date oops("Stardate 3734.0");
```

因此，程序库的设计者应该让 Date 的构造函数测试给定的字符串是否有效这个前提条件。下面是我们检测前提条件的代码：

```
Date::Date(const char* s) {
    if(!valid_date_string(s)) {
        //前提条件测试失败——产生错误!
        //...
    }
    //...
}
```

上面代码中，valid_date_string 函数在给定的字符串可以有效地解释为日期的条件下，就返回真值。因此，对所有的函数检查所有的前提条件是检测许多错误的有效方式。

5.2.2　表示不变性

另一种捕获错误的不变性是表示不变性。一个类的表示不变性是指下面的特性：对于这个类的任何合法的或者合理的对象，这个特性总是能成立的。例如，假设我们的 Date 类是通过下面的两个函数来实现的：

```
class Date {
private:
    int jdn;    //  代表儒略日（罗马时期的一种日期形式）的数字
    int dow;    //  星期几
public:
    //...
};
```

成员 jdn 是代表儒略日的数字——即从公历 4711 年 11 月 24 日始到现在流逝的天数[1]；成员 dow 的值域为从 0 到 6，它表示星期几。虽然星期几可以通过儒略日数字计算出来，但这种计算是非常复杂的；因此预先储存 dow 成员，用于提高执行期效率。这里的 Date 类的一个表示不变性就是：dow 的值等于 jdn 的值所对应的星期几。

某些程序库的设计者给每个类添加一个检测函数，用于检测这个类的一个或多个表示不变性：

```
class Date {
public:
    bool ok() const {
        return dow == jdn2dow(jdn); //这就是本讨论的表示不变性。
    }
    //...
};
```

上面的代码中，jdn2dow 函数执行复杂的运算，返回所给儒略日数字所对应的星期天数（即星期几）。

为了避免产生误解，我们应该清楚：一个类的不变性并不一定是类中每个成员函数的前提条件，即有些函数可以不包含这些不变性（前提条件）。假设 Date 类含有下面两个操作：

```
class Date {
private:
    void incr_dow();
public:
    void operator++();
    //...
};
```

其中，函数 incr_dow 逻辑地（即加到 6 以后又返回 0）递增星期数：

```
void Date::incr_dow() {
    dow = (dow + 1) % 7;
}
```

函数 operator++递增日期：

```
void Date::operator++() {
    ++jdn;
    incr_dow();
}
```

注意，当 incr_dow 被 operator++调用的时候，ok 检查的不变性明显不能满足。因此，这个不变性并不是 incr_dow 函数的前提条件。（如果我们让 operator++在递增 jdn 之前调用 incr_dow 函数，那么 incr_dow 函数也就能满足 ok 函数中的不变性。）

在特殊情况下，如果某个成员函数是被类的用户（不是类的函数）调用，那么表示不变性就必须能够成立。因此，我们愿意编写下面的代码：

[1] 对阳历日期使用修饰符 B、C、和 A、D 有些让人难懂，因为在 16 世纪以前阳历（新式历法）不存在，而儒略历（旧式历法）是存在的，但是没有第 0 年。如果把阳历后推并假定阳历 0 年存在，那么阳历年－4711 就是上述数字表示的年份。

```
void Date::incr_dow() {
    if(called_by_user_of_Date() && !ok()) {
        //产生错误时执行的代码。
        //...
    }
    //...
}
```

然而，在调用成员函数的时候，要判断究竟是谁调用了这个函数的做法是很不现实的。因此，无论是谁调用，每个成员函数都加入对所有函数都适用的、最严格的前提条件。

5.3　处理错误

假设我们需要检查程序库代码中的某个错误；接下来，我们还必须决定程序库在错误出现的时候，如何处理错误。很显然，没有对所有错误都适合的处理方法。处理错误最好的办法要取决于错误的性质和当错误产生的时候，代码执行哪些处理操作。另外，对于处理错误，程序库不同的变量可能会有不同的处理错误的方法。在这一节里，我们将讨论程序库用于响应错误所执行的最普遍的行为。

5.3.1　程序库变量

程序库可以在不同的时间，由不同的人，为不同的用途而被执行。例如，程序库的开发者执行程序库可能是为了测试它的功能或者衡量它的性能；程序库用户执行程序库一般是在开发他们自己程序代码的时候，或者他们自己或他们的用户运行已开发应用程序的时候。

为了满足这些不同的需要，程序库应该具有几种不同的变量：开发变量（development variant）用于开发和调试程序库本身；调试变量（debugging variant）用于开发和调试用户代码；优化变量用于获得高效的执行效率。

程序库变量并不需要具有完全相同的行为。特别地，当所给的错误出现的时候，它们（程序库变量）可以具有不同的行为。对于开发变量和调试变量，我们总是希望能够尽可能多地检测出错误，并且可以尽早地检测出错误。而对于优化变量，错误检测常常为提高速度而遭到忽视。

实际上，程序库中开发变量和调试变量的行为经常是相同的，因为程序库的实现者和用户在他们开发和调试各自代码的时候，往往都需要执行相同的检验。

5.3.2　解决问题

有时，当错误出现的时候，我们可以解决这个问题（暂时改正这个错误），并让这个程序继续执行，好像这个错误并没发生似的。但是，请先考虑 5.2.2 小节给出的 Date 类，如果在 Date::ok 函数中，我们发现表示不变性是不成立的，那么我们就可以恢复这个表示不变性：

```
bool Date::ok() {
```

```
    int correct_dow = jdn2dow(jdn);
    if(dow != correct_dow) {
        dow = correct_dow;
        return false;
    }
    return true;
}
```

(注意：如果我们使用上面这个方法，那么函数 ok 将不再是 const 函数，因为它实际上改变了成员变量的值)。当然，恢复这个表示不变性并没有解决产生错误的最终根源，但这种做法可以使程序表现得更加健壮。

5.3.3 程序退出或者程序中止（Exit or Abort）

当错误发生的时候，程序库可以执行的最有力的措施就是退出或者中止程序。考虑 Date 的构造函数：

```
Date::Date(const char* s) {
    if(!valid_date_string(s)) {
        cerr<<"Date: invalid date string, aborting\n";
        abort();
    }
    //...
}
```

然而，对用户而言，退出或者中止程序通常是不可接受的。因此，许多 C++程序员要求他们使用的程序库永远也不会退出或者中止程序。

5.3.4 抛出异常

一个优雅的处理错误方式就是抛出异常：

```
class Date {
public:
    class Invalid { };
    //...
};
Date::Date(const char* s) {
    if(!valid_date_string(s))
        throw Date::Invalid();
    //...
}
```

Invalid 类是一个空类，它是给 Date 的构造函数抛出异常使用的。通过捕获这个异常，用户可以检查日期字符串的有效性：

```
void usedate(const char* s) {
    try {
        Date date(s);
        //...
```

```
        }
        catch (Date::Irvalid) {
            cerr<< s << "is not a valid date\n";
            //...
        }
        //...
    }
```

抛出异常也存在几点不足之处。首先，并不是所有的 C++编译器都支持如上面代码的异常处理机制；因此，任何抛出异常的程序库将不再具有系统可移植性，因为对某些系统而言，它们根本就不存在支持异常处理的 C++实现。其次，对某些系统而言，抛出异常会导致运行速度变慢。再次，比起那些不支持异常的编译器，某些支持异常的 C++编译器会产生许多既大又慢的目标代码——即使异常实际上并没有被抛出。为了避免下面这些开销，许多 C++程序员都选择使用不引入异常的代码。

某些 C++实现提供了一个异常的编译时选项，用它来关闭对异常处理的支持；但是，我们并不能依靠这个选项来补偿异常的性能问题。假设存在一个使用异常的程序库，当异常选项打开的时候，它是支持异常编译的。你可能会这样想：对于那些不需要捕获异常的用户，他们可以通过在编译时关闭异常处理，来避免他们代码中由于异常处理导致的开销。然而，对于那些只有部分翻译单元支持异常处理的程序，它们的行为往往是不确定的。这样的程序可能会链接失败，或者只有在没有异常抛出的时候，才能够正确链接并执行。对于这样的程序，当有异常抛出的时候，系统的 terminate 函数将会被调用，这样程序将会中止，或者导致某些不确定的行为，而我们唯一能够肯定只是：程序不会调用任何用户提供的异常处理程序（因为原来的异常处理已经被关闭了）。

异常的另一个缺陷是，把非局部的控制流程引入到程序中。我们将在 5.5 节讨论可能的非局部控制流程为可重用代码设计者带来的问题。

5.3.5　返回错误值

在异常被添加到 C++之前，一个处理错误的普遍方法就是返回错误值。例如，假设类 Date 含有实现年份递增的 incr_year 函数。于是，我们可以如下声明这个类：

```
class Date {
public:
    bool incr_year();
    //...
};
```

对于任何 Date（日期）对象，如果表示的不是某年的 2 月 29 号，那么 incr_year 函数将会递增这个对象的年份，并返回 true 值，否则 incr_year 将返回错误值 false。

返回错误值有几个不足之处。首先，某些函数（特例如构造函数和析构函数）不能返回错误值。另外某些函数则不能很便利地返回错误值。例如，假设类 Date 有一个 next_year 函数，它返回相对于给定日期一年后的日期：

```
class Date {
public:
    Date next_year() const;
    //...
};
```

基于上面的实现，没有任何日期可以被合理地解释为错误值。因此，为了从 next_year 返回一个错误值，我们可以改变函数的参数和返回类型：

```
class Date {
public:
    bool next_year(Date& d) const;
    //...
};
```

上面这个版本的 next_year 函数把参数 d 设值为下一年的日期，并返回一个标志是否错误的布尔值。然而，它的接口将比不上第一个函数优雅。（另外一种方法是，next_year 可以返回一个含有一个 nil（零）值的 Date 对象，我们将在下一小节讨论这种技术。）

返回错误值的第二个缺点是用户可能未能检查这些错误值。很明显，忘记检查错误值是错误的一个常见来源；所以，程序库应该避免使用那些容易产生错误的接口。

另外，如果一个程序中的许多函数都返回错误值，那些记得要检查所有错误值的用户就不得不编写检测代码，这样程序的主控制流程就淹没在为检查和传播错误值所需要的许许多多的代码之中了，这似乎有喧宾夺主之嫌。

返回错误代码的缺点促使了 C++添加了异常这个机制。用异常来取代返回错误，使程序员可以只在适当的地方检查错误，并且免除了所有致力于返回错误值的代码。

5.3.6 创建 Nil 值

某个错误如果是在构造函数执行的时候出现的，那么与其退出、中断、或者抛出异常，我们不如创建一个 nil 值：

```
class Date {
private:
    bool is_nil;
    //...
};
Date::Date(const char* s) {
    _is_nil = !valid_date_string(s);
    //...
}
```

当然，用户需要一种可以确定给定的对象是否为 nil 的方法：

```
class Date {
public:
    bool is_nil() const {
        return _is_nil;
    }
```

```
        //...
    };
```

对于上面这个设计，Date 的构造函数并没有加入给定字符串必须有效的前提条件

当我们把 nil 加到类的抽象的时候，对于那些包含一个或者多个 nil 对象的成员函数，我们必须规定这些函数的行为。例如，假设类 Date 有一个 dayno 函数，它返回从今年 1 月 1 号到现在流逝的天数：

```
class Date {
public:
    int dayno() const;
    //...
};
```

当给定的 Date 类含有 nil 成员的时候，我们必须规定 dayno 的行为；可能的行为在 5.3.2 小节到 5.3.8 小节都有介绍。例如，dayno 可以抛出一个异常等等。

5.3.7　把无效的数据解释为有效的数据

程序库设计者有时会通过这样方法来处理错误：把无效的数据解释为某些其他的有效数据。例如，在 Date 的构造函数中，我们可以把那些无效的日期字符串解释为 1970 年 1 月 1 号：

```
Date::Date(const char* s) {
    if(!valid_data_string(s))
        //把日期数据设为 1970 年 1 月 1 号
    //...
}
```

然而，这样的设计是很容易产生错误的。考虑计算贷款利息的 compute_interest 函数，并假设在输入中出现了一个打印错误，如：

```
July 21,19992
```

然而，这个错误将会使计算利息的开始日期比正确日期提前 30 年左右；除非某人在这条语句被运行之前，捕获了这个错误，否则借款人肯定会大吃一惊。与这种方法相比，当错误出现的时候，如果让 Date 的构造函数退出、中止、抛出一个异常或者创建一个 nil Date 对象，那么调试这个错误将会比较容易。

5.3.8　允许不确定的行为

某些时候，检测某个特殊的错误是非常低效的。假设目前我们需要检查 5.1 节 Stack::pop 函数的前提条件，看某些语句是否违背了这个前提条件：

```
template<class T>
T Stack<T>::pop() {
    if(size() == 0) {
        //error!
        //...
    }
```

```
    //...
}
```

现在考虑下面的用户代码：

```
Stack<int> s;
//...
assert(s.size() > 10000);
for(int i= 0; i < 1000; ++i) {
    int nextval = s.pop();
    //...
}
```

由上面代码可知，每次调用 pop 函数都会检查该函数的前提条件。然而，用户知道在上面循环中，该函数的前提条件是永远成立的。为了使用户能够编写高效的代码，我们可以只在程序库的调试变量（见 5.3.1 小节）里检查这个前提条件（或者所有的前提条件）。另一方面，如果其他一些变量违背了这个前提条件，那么错误将不能得到检测，于是用户的程序将会有不确定的行为。

有时，检测一个给定的错误是很困难甚至不可能的。假设我们的程序库提供了另外一个版本的全局操作符 new 和 delete；如果对对象 p 执行 delete 操作，那么前提条件是 p 值的空间是由之前调用 new 操作分配的。要高效并且可移植地检测这种前提条件是有可能办到的，但实现起来将会非常困难（见练习 5.2）。因此，对于程序库的任何变量，我们可以选择不检测这类错误，而允许包含这类错误的用户程序有不确定的行为。

5.4 资源限制（Resource-Limit）错误

在第 4 章中，我们讨论了程序库设计者所关心的许多资源问题；并且给出最小化使用这些资源的方法。在这一节里，我们将讨论，尽管设计者尽了最大努力，但是某个有限的资源已经用完，并且需要更多的这类资源，而且不能得到这类资源，此时程序库应该（或者根本就不应该）如何处理这类错误。

5.4.1 堆栈溢出

一个存在堆栈溢出的程序通常都有不确定的行为。例如，假设当调用下列函数的时候，系统只剩余 100 字节可利用的堆栈空间：

```
void f() {
    T t;
    //...
}
```

如果 sizeof(T)超过 100 字节，那么调用 f 函数将会有不确定的行为发生；在大多数系统，程序将会中止。因为不存在可移植的方法，用来决定栈是否有潜在的溢出危险，所以几乎不能检测这个问题，或者当这个问题发生的时候，几乎没有恢复系统的希望。如 4.5.4 小节所讨

论的一样，如果我们在空闲存储空间分配大的对象（即不在堆栈空间分配），那么堆栈溢出的机会将会减少。

5.4.2 用完空闲存储空间

如果空闲存储空间只有100字节的可利用空间，而sizeof(T)大于100字节，那么调用new T将会有下面的结果：对于那些支持ANSI/ISO的C++实现，将会返回0值，或者抛出一个ANSI/ISO C++中的异常（当默认的new_handler函数被调用的时候）。

和堆栈溢出类似，不存在可移植的方法可以用来判断空闲存储空间是否濒临用完（见练习5.3）。例如，如果没有足够的内存的话，new(实际上是new_handler)实现将会被中止。但是，如果new返回控制权给调用它的程序，那么某些时候我们却可以很健壮地处理空闲存储区的溢出。

一种预防空闲存储区溢出的方法就是尝试分配更小的空间。例如，如果我们希望分配一个很大的哈希表但是却不能成功；我们就可以对这个哈希表的大小减半，然后试着继续分配。如果使用符合ANSI/ISO标准的C++实现，我们可以如下编写代码：

```
int min_size = 1024;
int size = 1024 * 32;    // 第一次尝试分配32K。
Table_entry* table = new Table_entry[size];
while(table == 0 && size >= min_size) {
    size /= 2;
    Table = new Table_entry[size];
}
```

这里，如果分配不能成功，我们就减半哈希表的大小，并试着继续分配，直到减半后的哈希表大小足够小，可以分配为止——甚至有可能比min_size还要小，但一定比min_size的一半大。

另一种可以健壮解决空闲存储区溢出的方法是删除存储区的某些东西，然后继续尝试分配：

```
T* t = new T;
while(t == 0) {
    delete something;
    t = new T;
}
```

当然，如果程序库实现者采纳4.5.2小节的建议，尽快地删除不需要的对象，那么要找到可以删除的东西就变得很困难了。在一般情况下，高速缓存里面的东西是可以删除的。例如，假设为了效率因素，我们的程序库在内存中有给定文件内容的高速缓存；如果空闲存储区溢出，那么我们就可以删除这些高速缓存的内容，并尝试继续分配。

通常，当空闲存储区溢出的时候，程序库最合理的解决方法就是抛出一个异常。但是，如果用户的C++实现不支持异常，那么我们唯一的办法就只能是中止程序：

```
T* t = new T;
if(t == 0) {
    cerr << "Library Q: out of memory \n";
```

```
        abort();
    }
```

然而，我们发现如果流插入语句（cerr<<）的执行需要分配内存空间，而并没有所需要的内存空间可以分配时，那我们该如何是好呢？一种用于避免这个问题的方法就是保留一小块内存——用来处理程序内存分配失败时使用的内存，这块内存的大小足够完成我们需要的清理工作。当我们发现空闲存储空间已经被用完的时候，就可以释放这块预先保留的内存块，并继续我们的错误处理操作。这个做法也是有缺点的，如果每个程序库都保留了这么一块用于错误处理的内存块，那么对于一个使用了好几个程序库的程序，它将会有很大一部分的内存不得不用于防止不太可能发生的事件（即错误）。

在上面的例子里，对于每个调用 new——而不是调用 set_new_handler——的返回值，我们给出了检查这些返回值的程序库代码（将在 6.5 节看到，C++程序库不应该直接调用 set_new_handler）。然而，在 ANSI/ISO C++中，每次检查 new 调用是没有必要的。如果希望每个失败的 new 调用都抛出一个异常（实际上这都是我们通常所希望的），那么就不能只是捕获这个失败的 new 抛出的异常，：

```
T* t = new T;
//如果空闲存储区溢出，抛出一个 ANSI/ISO C++异常
//于是，这里的代码可以假设 t != 0.
```

如果我们希望更加健壮地处理这个错误，可以捕获这个异常：

```
T* t = 0;
try {
    t = new T;
}
catch (bad_alloc) {
    delete something;
}
```

这个 catch 捕获的异常可能是前面 T 的构造函数调用 new 后间接抛出的异常。因此，程序库的实现者必须能够保证：无论是哪个 new 抛出的异常，代码都可以正确地处理这些异常。

5.4.3 文件系统限制

如果用户机器有一个文件系统，而且文件系统已经用完了所有的空间，那么任何希望增加文件的操作都将失败。例如：

```
ofstream f("tmp");
f << "Make it so!";
if(!f) {
    cerr << "write to file failed \n";
}
```

然而，在一个程序里面，我们很难分辨输出错误的下面两种原因：究竟是文件系统溢出造成输出错误，还是某些其他的问题（如网络连接失败等等）导致了输出错误。如果不知道输出操作错误的原因，那么要想编写出健壮的预防代码将是非常困难的。我们可以通过尝试

删除某个文件以释放某些存储空间，然后再次尝试写入操作；但这些操作只有在错误是文件系统溢出造成的情况下，才能起作用。因此，通常最合理的解决方法就是抛出一个异常、退出程序或者中止程序。

在给定时刻，某些系统限制一个程序可以打开的文件数目，通常情况下是 32 或者 64。如果一个程序打开的文件达到了最大数目，那么企图打开另一个文件的操作就会失败：

```
ofstream f("tmp");
if (!f){
        //打开的操作失败
}
```

通过 iostream 接口来判断企图打开文件的操作是否失败是不可能的，因为打开的文件实在太多了。虽然大多数系统都提供了系统调用，让我们可以找出错误的原因，但是我们往往都不会在乎这些文件打开错误的原因——因为所有的程序库都可以智能地抛出一个异常、退出程序或者中止程序。

5.5　异常安全性

在介绍 C++语言的异常处理的同时，可重用代码的编写者必须确定他们的代码是异常安全的——就是说，当异常被抛出的时候，他们的代码仍然可以正常地运行。对于现今存在的在没有异常抛出的环境下，可以正确运行的代码，当我们需要把这些代码移植到具有异常处理机制的环境时，就可能需要修改这部分代码。

因为异常可能引入非局部的控制流程，所以代码中往往就会出现问题。当一个异常被抛出时，堆栈是一直展开的，直至找到了异常的处理语句。因此，当一个函数调用另一个函数的时候，控制权就有可能不会返回到调用的函数。例如，对于包含有空异常抛出的函数 g，如果函数 f 调用了函数 g，那么这个调用就不一定是安全的：

```
void g() throw();
```

某个异常可能会传递到 g 函数中，此时控制权将会转移到不确定的（unexpected）的函数，而不会返回到 f 函数。

显式的函数调用也不是异常的唯一来源。实际上，编译器为实现某些表达式而自己产生的代码也可以调用某些函数（比较典型的例子有构造函数、析构函数、new 和 delete），这些函数也会抛出异常。例如，基于 ANSI/ISO C++标准，当 new 操作失败时，new_handler 将会抛出一个异常。

如果在一段代码中可能有许多地方会出问题，我们就称这段代码是异常不安全的。5.5.1 小节和 5.5.2 小节将会讨论两种潜在的这类问题。

5.5.1　不一致的状态

成员函数的实现通常把临时对象看成是不一致的状态。我们来看 5.2.2 小节给出了一个例

子，在函数 Date::operator++的实现中：

```
void Date::operator++() {
    ++jdn;          //行 1。
    incr_dow(); //行 2。
}
```

如果在调用 operator++的时候，*this 是一致的，那么在刚调用完行 1 后，*this 就变成不一致的；要直到调用完行 2 之后，*this 才又变成一致的。

假设当*this 不一致的时候，被调用函数会（直接或者间接地）抛出一个异常：

```
void Date::operator++() {
    ++jdn;
    might_throw_exception();
    incr_dow();
}
```

如果 might_throw_exception 抛出了一个异常，那么将会留下一个不一致的*this。即使类 Date 的用户捕获并且处理了这个被抛出的异常，仍然会留下一个不一致的对象。考虑下面的用户代码：

```
void f(Date& d) {
    try {
        //...
        ++d;
        //...
    }
    catch(Some_exception e) {
        //...
    }
```

如果 Date::operator++中调用的 might_throw_exception 抛出一个类型为 Some_exception 的值（异常），那么当 f 函数返回的时候，d 引用的对象将会是不一致的。

如果不可能出现这种情况——在类 X 的任何成员函数执行期间，这个函数抛出了一个异常，给类 X 的用户留下了一个不一致的 X 对象——那么类 X 就是异常安全的。可重用的类应该尽可能是异常安全的。为了增加类 Date 的异常安全性，我们可以把 might_throw_exception 调用移到可以令*this 保持一致性的位置：

```
void Date::operator++() {
    ++jdn;
    incr_dow();
    might_throw_exception();
}
```

显然，这个转移能否保持类 Date 的正确性要取决于各种相关函数的期望行为。

实际上，几乎所有的函数都可能会抛出异常；把所有上面这类函数的调用都转移到可以令*this 保持一致的地方是很困难的，或者说，要想高效地实现这种转移是很困难的。假设我们不能把 might_throw_exception 调用转移到使*this 保持一致的地方，那么如果需要保持 Date

类的异常安全性，那么就必须如下编写 operator++代码：

```
void Date::operator++() {
    ++jdn;
    try {
        might_throw_exception();
    }
    catch(...) {
        --jdn;
        throw;  //再次抛出异常。
    }
    incr_dow();
}
```

初看起来，上面的代码比较不好实现并且是难以理解的。实际上，要实现一个异常安全的类是很有挑战性的，有时候甚至是不切实际的。

如果某个成员函数不是异常安全的，那么这个函数的文档和相应的类都应该有如下说明：

这个函数[或者类]不是异常安全的。

由于异常是一个比较新的 C++特性，并且要实现一个既高效又异常安全的类是困难的；所以现今使用的许多类都不是异常安全的；而且大多数非异常安全的类至今还没有给出如上的说明文档。

5.5.2　资源泄漏

在 C 和 C++中，一个传统的编码风格就是：先分配或者锁定某些资源，然后使用这些资源执行某些计算，最后释放这些资源。遗憾的是，这个编码风格并不是类型安全的。考虑下面的代码：

```
void f() {
    Widget* w = new Widget;
    g(w);
    delete w;
}
```

如果有一个异常从 g 函数抛出，那么 w 指向的已经分配内存的对象将永远得不到释放。

有两种方法可以用来避免这种资源泄漏，并且增加代码的异常安全性。第一，我们可以在资源需要释放，或者有可能抛出异常的地方，使用一个 try 块和一个 catch 子句：

```
void f() {
    Widget* w = new Widget;
    try {
        g(w);
    }
    catch(...) {
        delete w;
        throw;      //再次抛出一个异常。
    }
```

```
    delete w;
}
```

上面这种编码风格的缺点是，如果把它完全应用到现实函数中去，那么将会导致可读性很差的 try 块网状结构。一个更加简洁的解决方法就是：在类的构造函数中分配资源，并在析构函数中释放资源——就是著名的“初始化中获得资源”技术[ES90,Str91]。为了把这种技术应用到空闲存储区对象，我们可以编写下面的类：

```
template<class T>
class New {
public:
    New() : t(new T) { }
     New() { delete t; }
    operator T*() { return t; }
private:
    T* t;
    //隐藏这些成员。
    New(const New& n);
    const New& operator=(const New& n);
};
```

我们有意隐藏了拷贝构造函数和 New 的赋值运算符，因为定义它们的语义在这里将会比较困难，况且我们也不需要它们（见 2.4 节）。

使用上面的 New，可以如下编写我们的函数，而不需要使用 try 块：

```
void f() {
    New<Widget> w;
    g(w);
}
```

这样，当 w 对象被删除的时候，所分配的对象也会被释放，而无论 g 函数是否抛出一个异常。

5.6 总结

用于重用的代码必须考虑，是否检测错误和当错误出现的时候，如何处理错误。不变性可以用于检测许多种错误；程序库也应该很好地利用函数前提条件和表示不变性来为检测错误服务。

程序库各种不同的变量可以用来处理各种不同的错误。下面是处理错误最普遍的方式：

- 纠正问题（暂时解决错误）并让程序继续执行；
- 退出程序或者中止程序（对许多程序库而言，这种方法不可行）；
- 抛出一个异常；
- 创建一个 nil 值；
- 把无效的数据解释为有效的数据；

- 根本不检测错误（因此也会导致不确定的行为）。

在所有的错误中，程序库设计者还会遇到系统资源用完的错误。我们给出了三种可能情况：堆栈溢出，空闲存储空间被用完，某些文件系统的空间和打开的文件数达到最大限制值。

随着异常被引入 C++语言，程序员就必须特别小心，以使确保所编写的可重用代码是异常安全的。对于类的设计，当异常被抛出的时候，程序应该避免出现非一致性的对象。对于程序库设计，当异常被抛出的时候，程序库应该避免因非局部控制流程而导致的一些负面效应。

5.7　练习

5.1　在 5.2.2 小节，我们认为：类的不变性并不一定是类中每个成员函数的前提条件；然而，当这个成员函数是被类的用户调用时，这个不变性的值必须成立。假设我们为了判断类 X 的成员函数是否是被该类的用户调用（相对于被类 X 的其他成员函数调用），我们在每个成员函数的入口处都测试一个计数器；每个成员函数在测试这个计数器之后，都会自动增加计数器的值，而当函数返回之前的那一瞬间，都会减少计数器的值。这个设计方案的目的在于，当成员函数是被用户调用而不是被其他成员函数调用的时候，这个成员函数就可以很自然地检查类的不变性。

a. 对于虚函数，这个设计方案是否可以行之有效呢？对静态函数呢？对那些具有静态成员的类呢？

b. 为了达到这个目的：无论 X 的某个成员函数是被非 X 的成员函数调用，还是被 X 的成员函数调用，在这两种情况下，这个函数都可以检查类的不变性，你要如何扩展上面的设计呢？

c. 这个设计是异常安全的吗？

d. 为了检查类的不变性而实现这样的设计方案，你认为这种做法值得吗？

5.2　在 5.3.8 小节，我们认为 delete 有这样的前提条件：它的参数值必须是之前调用 new 而得到的。

a.（*）给出实现具有可移植性的 delete 和 new 版本的方法，从而让 delete 可以检查是否符合该前提条件。

b. 讨论 a 部分解决方法的效率问题。

5.3　在 5.4.2 小节，我们曾说过：没有任何完全可移植的方法，用来判断空闲存储区是否已经接近用完。然而，对于濒临用完的空闲存储区，在某些系统上是可以探测得到的。

a. 编写一个给系统使用的函数 freestore_avail，当用户需求的内存空间大于可利用的空间的时候，这个函数就返回 0 值。在通常情况下，这个函数应该返回小于或者等于可用空闲存储空间字节数的 2 的最高次幂的值（例如，假设只有 250 字节的可利用空间，那么 freestore_avail 函数将返回 128）；当没有可利用空间的时候，这个函数将返回 0 值。

b. 假设 freestore_avail 函数调用表明你所需要的内存数量是可以从系统得到的，并且你尝试了分配这些内存空间。然而，内存空间分配却失败了，这是为什么呢？

5.4 假设你在实现一个名为 Compact_tree 的类，它描述一棵以紧形式存储的树。这棵树并没有使用指针，因为指针会占用很大的空间；你决定使用数组来实现 Compact_tree 类，其中数组成员为树中前序遍历的节点：

```
class Compact_tree {
private:
    Node* array_of_node_in_preorder;
    //...
};
```

你还使用了深度优先查找，用树中每个节点在查找中的序号来描述每个节点：

```
class Node {
private:
    int dfsnum;
    //...
};
```

并且假设类 Compact_tree 存在一个成员函数 advance：

```
class Compact_tree {
public:
    bool advance(Node& n);
    //...
};
```

如果 n 有一个前序遍历的后继者，那么函数 advance 将把 n 设置为指向这个后继者的引用，并返回 true 值；否则函数 advance 将返回 false，且对 n 没有任何影响。

a. advance 函数的前提条件是：n 引用的是调用 advance 函数的 Compact_tree 中的某个节点。为了使 advance 能够检查这个前提条件，你需要在 Node 中增加哪些成员变量呢？

b. 假设只有当宏 NDEBUG 没有定义的时候，才需要添加刚才增加的成员变量。为了使 Node 对象能够尽可能地小，你应该如下声明成员变量：

```
class Node {
private:
#ifndef NDEBUG
    //你的成员变量应该在这里声明。
#endif
    //...
};
```

如果你是如上面有条件地声明你的成员变量，那么用户碰巧需要链接某些声明有 NDEBUG 的代码和其他一些没有声明 NDEBUG 的代码，则会发生什么情况呢？

c. 既然编译系统并没有检查是否有违反 C++的“一次定义规则”的操作，那么你认为检查 advance 的前提条件还是个很好的主意吗？

5.5 考虑下面的代码：

```
clock_t clock1 = clock();
time_consuming_computation();
clock_t clock2 = clock();
cout << "time_consuming_computation took approximately" <<
    (clock2 - clock1)/CLK_TCK << " seconds to complete \n";
```

编写这段代码的目的在于：报告耗时的计算（time-consuming computation）究竟需要持续了多长时间。但是这段代码存在一个错误：如果由于某种原因，调用 clock()并不能得到已流逝的处理器时间，那么这个函数将会返回-1。

a. 修改上面的代码，从而当 clock（）调用失败的时候，不会报告错误的结果。

b. 假设 clock_t 是在 32 位的机器上实现的，而且 CLK_TCK 的值为 1000000；那么 clock 返回的时间大约只能是在 0 到第 36 分钟范围以内（这是由 Harbison 和 Steele 在[HS91]提出的）。修改上面的代码，以便当 clock 在上面范围回绕（即超过最大值后又返回 0）的时候，不会出现错误的结果。

c.（**）当 clock 回绕多于一次的时候，b 部分的解决方案可能已经不能得到正确的结果。是否可以修改你的代码来达到下面的目的：无论时间有多长，最后报告的结果总是和 time_consuming_computation 计算的结果完全相符。

d. 你在 b 部分和 c 部分的解决方案可能都假设了：无论何时调用 clock，它返回的值都会比第一次调用 clock 返回的值小，这是因为时钟发生了回绕。假设在你的两次调用 clock() 函数之间，某人由于某种原因重置了处理器时钟，那么如果要区分：时钟究竟是被重置了，还是发生了回绕，将会有多大的困难呢？时钟被重置的可能性大概有多大呢？程序库是否应该检测这种错误呢？

5.6　有时，人们会讨论关于如何使代码达到最大化健壮的问题。最大化健壮的代码应该检测所有可能出现的错误。考虑下面的代码（并不是最大化健壮的代码）：

```
ofstream f("tem");
f.write("hello",5);
f.flush();
ifstream g("tmp");
char buf[5];
g.read(buf,5);
```

上面的代码首先打开一个名为“tmp”的文件，如果这个文件不存在的话，将新建这个文件。打开流 f 是为了写入内容并且初始定位在文件的开头。代码接着写入 5 个字符到文件中，并且刷新流，以确保字符确实已经写入文件中了。然后，代码打开同样的文件，但这次是读文件，并读出刚才写入的 5 个字符。

a. 什么样的系统错误会导致上面的代码运行失败呢？

b. 为了得到代码的最大化健壮性，我们应该如何改写上面的代码呢？（不必要给出每个可能的错误的处理方法。）

c. 对于许多软件开发项目，检查每一步可能失败的操作往往是得不偿失的。而且，对于每一步可能导致失败的操作，要决定如何处理也是很困难的。如果你在编写包括上述代码段

的可重用代码，那么对于前面的代码，你会测试哪些可能导致失败的操作？而哪些操作你将不会检查呢？

5.7 在第4章，我们讨论了最小化程序的创建时间、运行时间、代码大小和内存占用量。然而，这些并不是程序库必须高效使用的资源；如5.4.3小节讨论的那样，大多数系统会限制程序的一定时间内可以打开的文件的最大数目。因此文件打开数目也应该有一定的节制。

考虑在4.5.1小节的preprocess函数中处理#include语句的代码。当preprocess在它的处理文本中碰到一个#include语句的时候，它需要打开这个被引用的文件，并且预处理这个文件的内容。

a. 一个自然的用于实现preprocess的策略是：维持一个嵌套的#include堆栈，堆栈中的所有文件都处于打开状态。假设某个用户系统允许32个文件同时打开，并且在调用preprocess函数来预处理代码之前，系统已经有31个文件处于打开状态了。如这个预处理代码过程需要#include具有3层深的嵌套。那么，如果preprocess没有检测到这种导致的错误，将会出现什么情况呢？你是否可以想出某种既可以处理这个错误，又可以令用户满意的方法？

b. 你是如何让preprocess可以处理嵌套#include的，而无论用户系统支持的嵌套层次有多深，即使用户是在所有文件已经被打开的时候调用preprocess也可以处理？

5.8 给出两个理由说明：当5.5.1小节的Date::operator++被调用的时候，*this本身可能是不一致的。

5.8 参考文献和相关资料

介绍程序正确性理论的好书可以在下面找到：Stanat和McAllister[SM77]的第1章、Meyer[Meyer88]的第7章和Liskov与Guttag的[LG86]。Meyer在[Meyer92a]的第9章讨论了不变性和前提条件。

Dershowitz与Reingold的[DR90]和Reingold、Dershowitz与Clamen的[RDC93]都讨论了儒略日期数字、阳历和其他日历体系的内容。

编写与文件系统交互的健壮代码是非常困难的。那些致力于健壮使用iostream库的C++程序员应该阅读Teale的[Tea93]或者Plauger的[Pla95]。

Cargill在[Car94]详细讨论了编写异常安全类的难度。

对于检测C++程序的资源泄漏，现今有几个商业工具可以使用。

练习5.4的Compact_tree和Node类的设计灵感来自于Koenig在[Koe92c]关于相似类的描述。练习5.5的设计灵感来自于Harbison和Steele在[HS91]的某个例子。

第6章 冲突

又一次把我们带到了一起。
——Richard Nixon
所有的问题都是针对某一个名字空间而言的。
——Jerry Schwarz

对于两部分软件，如果它们不能在同一程序里很容易地用在一起，那么我们就称这两部分软件是冲突的。可重用代码应该尽可能地避免这种冲突。在同一个作用域内，可能会使用相同的名字来代表不同的事物，而许多代码的冲突就与这密切相关。程序库使用的名字可能会和其他程序库（或者用户代码）的名字发生冲突。这些冲突包括全局名称、宏名称和环境名称相互之间的冲突。我们将在这一章中讨论各种冲突和描述命名约定（naming convention）与名字空间结构（namespace construct）的使用，并用它们来避免产生冲突。我们还给出了unclean 程序库的概念和 good-citizen 程序库的概念。

6.1 全局名称

C++程序里的所有代码——无论是用户代码还是程序库代码——都共享同一个（也只有一个）全局作用域。因此，程序库设计者应该非常地小心，避免在全局作用域定义的名字和其他程序库（或者用户代码）的全局名称发生冲突。例如，考虑两个程序库，每个程序库都定义了一个全局 Widget 类；因为对于同一个 C++程序而言，在同一个作用域内用相同的名称定义两个不同的类是非法的[1]，这样的两个程序库将会发生冲突，不能在同一个程序中一起使

[1] 实际规则要稍微复杂一些，在 6.1.2 节中我们可以看到这一点。

用。然而，并非所有代表不同事物的名字都会导致冲突。因此，为了理解冲突何时会存在，我们必须先仔细考虑C++程序的组织形式和编译方式。

6.1.1 翻译单元

假设a.h文件和a.c文件包含下列代码：

```
a.h:
    int i;
    #define greeting "Hey, Guys and Gals!"
a.c:
    #include<a.h>
    int main() {
        char* s = greeting;
        //...
    }
```

预处理a.c后的得到的翻译单元是（假设a.c的#include语句的内容指的就是上面的a.h）：

```
    int i;
    int main() {
        char* s = "Hey, Guys and Gals!";
        //...
    }
```

对于每个包含一个或者数个翻译单元的C++程序，程序中所有的翻译单元都共享同一个全局作用域。如果下面的两个翻译单元在同一个程序，那么i、j和k的定义都将在程序的全局作用域（有且仅有一个）：

```
unit1:
    int i;
    int j;
    //...
unit2:
    int k;
    //...
```

6.1.2 类的定义

关于什么样的声明和定义才能合法地出现在程序的全局作用域里，C++有规定这方面内容的许多规则。让我们先考虑类的定义。对于任何翻译单元，如果它包含了给定名称的类的多个定义（无论这些类的定义相同与否），那么任何包含这个翻译单元的程序都是非法的：

```
unit3:
    class X { /* 内容 */ };
    //...
    class X { /*相同或者不同的内容*/ };          //这样是非法的。
```

[1] 实际规则要稍微复杂一些，在6.1.2节中我们已可以看到这一点。

C++的编译器将能检测到这个错误。

现在考虑下面的两个翻译单元，每个翻译单元都包含了单一的名称为X的类的定义：

```
unit4:
    class X { /*...*/ };
    //...
unit5:
    class X { /*...*/ };
    //...
```

每一个包含上面两个翻译单元的程序可能是（也可能不是）合法的。如果上面两个类X的定义是相同的，并且其他地方不出现错误，那么这个程序就是合法的。如果X在两个翻译单元中的定义是不同的，那么这个包含程序的合法性要取决于类定义的链接形式。任何一个在给定翻译单元上定义的全局作用域类，如果在其他的翻译单元中，某段别的代码的声明使这两个翻译单元互相关联，那么就称这个类是外部链接的（请见后面的代码，当这本书出版的时候，ANSI/ISO 委员会正在考虑：某个标识符和其他标识符在某个声明上发生关联的定义）。上面的两个X中，如果一个或者两个X的定义是内部链接的（即不是外部链接的），并且其他地方不发生错误，那么这个程序就是合法的。

然而，如果两个X的定义都是外部链接的，那么包含这两个翻译单元的程序就是非法的。这个错误可能可以（也可能不可以）被现今的编译系统所检测。例如，假设每个类都包含了一个非内联的成员函数f：

```
unit4:
        class X {
            void f();
            //stuff
        };
    unit5:
        class X {
            void f();
            //不同的 stuff
        };
```

那么将会有两个不同的X::f()的定义，如果把这两个函数都传给链接器，那么链接器将会检测到这个错误。(关于链接器检测函数多种定义的条件的内容将在6.1.3小节讨论。)然而，如果X在两个翻译单元的定义具有足够大的差异，那么现今的编译系统将不能检测到这个错误。例如，假设两个X都只包含非静态数据成员（不包含成员函数）：

```
unit4:
        class X {
        public:
            int i;
            char c;
        };
    unit5:
        class X {
```

```
        public:
            int i;
            double d;
        };
```

那么现今的C++编译系统可能不能检测到这种错误。因此，如果包含着两个类的程序被创建并且执行，那么这个程序的行为将是不确定的。

6.1.3 函数和数据的定义

现在考虑函数的定义。（除了参数类型不同之外，数据定义和函数的定义是相似的。）对于具有同样的函数名和签名（包括参数的个数、顺序和类型）但有多个定义的翻译单元，任何包含这个翻译单元的程序都是非法的。例如，包含下面翻译单元的程序就是非法的：

```
unit6:
    void f(int) { /* 代码体 */ }
    //...
    void f(int) { /* 与上面相同的或者不同的代码体。 */ } //错误
```

这个错误可以被C++的编译器检测到。

如果相同函数的定义在不同的翻译单元，那么合法性将取决于定义的链接。如果一个或者两个函数是内部链接的（就是说，一个或者两个函数是用static关键字声明的）：

```
unit7:
        void f(int) {
            //...
        }
    unit8:
        static void f(int) {
            //...
        }
```

并假设不存在其他任何错误，那么这个程序就是合法的。

然而，如果这两个函数都是外部链接的，那么包含这两个翻译单元的程序就是非法的。这种错误能否被编译系统检测到要取决于这些翻译单元提交到链接器的方式。如果把unit7和unit8作为目标文件提交，那么链接器将会给出名为“多次定义”的错误。

然而，如果其中一个翻译单元——如unit7——被当作是目标代码档案文件的一个成员提交，那么这种错误能否被编译系统检测到将取决于其余代码的组织形式。而且，链接器可能不会把unit7包含进程序里面，虽然这样的结果文件是合法的，但是那些应该调用unit7中的f函数的程序，现在将会改而调用unit8中的f函数。所以，这个程序的行为是不妥当的。

6.1.4 程序库的蕴涵意义

现在让我们来考虑下面的情况：有关C++程序库设计的声明和定义C++规则的蕴涵意义。

假设某个程序库在全局作用域内定义了一个类 Widget，并且这个类是外部链接类型的[1]；那么，对于所有在全局作用域内，并且是以外部链接方式定义类 Widget 的其他程序库（或者用户代码），将会和上面的程序库发生冲突。对于那些希望在同一个程序内使用这样两个程序库的程序员，可能会在编译时得到一个错误，错误的出现也可能是在链接时，或者是程序库可以运行，但会出现不确定的行为。而且，即使在一个或者两个程序库中，Widget 的定义和使用只局限于该程序库内部，情况也是一样的。如果下面的文件：

```
libimpl.c
    // 只在这个文件内部使用，但可以有外部链接。
    class Widget {
        //...
    };
    //...
```

是程序库实现的一部分，那么任何包含 libimpl.c 目标代码的程序，都会把上面的 Widget 看作全局作用域类。

现在假设程序库定义了一个函数 f(int)。如果这个函数是内部链接的，并且是在（程序）库实现文件中定义的，那么就不会有发生冲突的可能性。否则，这个程序库将会和其他定义有外部 f(int)的程序库（或者用户代码）发生冲突；而任何希望在同一个程序中使用这两个程序库的程序员，都有可能会在链接时得到一个错误，或者只能是创建了一个具有不正确行为的程序。

全局名称冲突不但有可能使创建的程序具有不确定或不正确的行为，而且对于程序库用户而言，这些冲突还是很难解决的。为了削除一个冲突，这个程序库的预期用户必须修改程序库中一处或者两处的源代码。然而，修改源代码的操作是很乏味的，而且常常还有给别的地方引入错误；况且，如果用户没有源代码的存取权限，那么这将是不可能的。因此，C++程序库的设计者应该尽可能地避免全局名称冲突。为了避免全局名称冲突，C++程序库应该使用 namespace（名字空间）结构（除非由于可移植性问题而禁止使用名字空间，见 9.2.2 小节），而在名字空间不能使用的地方，命名应该符合命名约定。

6.1.5 命名约定

在所有代码的所有全局名称中，每个 C++程序库定义的、具有外部链接的全局名称都应该是唯一的，因为有可能会在同一个程序里面使用这些名称，从而导致冲突。为了提高名称惟一的可能性，每个具有外部链接的全局名称都应该加上前缀。在实际中，一个字母或者两个字母的前缀是很常见的：

```
class XYwidget {
    //...
```

[1] 更准确地说，是假设这两个条件至少有一个成立：（1）程序库中的头文件包含的代码导致了一个或多个用户翻译单元包含具有外部连接且作用域为全局的类 Widget 的定义。（2）程序库实现中的某个翻译单元包含了具有外部链接且作用域为全局的类 Widget 的定义。在下面的讨论中，我们将使用不太准确的语言。

```
};
void XYf(int);
```

为了进一步提高程序库名称惟一的可能性，我们还可以在前缀中加上程序库的名称，甚至开发这个程序库的公司的名称：

```
class Companyxyz_libabc_widget {
    //...
};
void companyxyz_libabc_f(int);
```

缺点是这样的名称不太好看，并且书写也有些麻烦。

如果程序库提供者愿意，他们可以给用户提供一个名称替换机制。通过提供一个替换名称的文件，用户可以随意地用他们喜欢的名字给文件命名：

synonyms.h:

```
#define Widget Companyxyz_libabc_widget
#define f Companyxyz_libabc_f
```

然而，使用预处理器来替换名称是有些风险的，考虑下面的用户代码：

```
#include<synonyms.h>
class User_class {
    class Widget {  //whoops!
        //...
    };
    //...
};
```

这里，用户的标识符（Widget）将会用库类名进行被替换——而这几乎不是用户所想要发生的。用户如果很幸运，这样的偶然替换应该会在编译时或者链接时出现错误；否则，我们将不得不对程序进行接二连三地调试。

程序库应该对所有具有外部链接的全局名称都应该应用命名约定——即使是那些只在程序库实现里出现的名称，也应该应用命名约定：

libimpl.c:

```
//具有外部链接。
class Companyxyz_libabc_used_only_in_this_file {
    //...
};
//...
```

如果程序库的实现非常地大，那么程序库的两个不同的实现者，可能会偶然地选择两个相同的名称。为了减少冲突的可能性，实现中的全局名称也应该把子系统名称嵌入进去：

```
libimpl.c:
    class Companyxyz_libabc_subsystemq_used_only_in_this_file {
        //...
    }
    //...
```

从理论上讲，如果不对类的私有成员名称进行保护，那么也会导致冲突。假设某个程序

库中的类X声明了一个私有函数f，而且，某些其他程序库也提供了类Y：

```
class Y {
public:
    void f();
    //...
};
```

另外，假设两个程序的用户进行从X和Y的多重继承，如下所示：

```
class XY: public X, public Y {
    //...
};
```

如果用户希望调用XY对象的f函数，那么将会出现一个编译时错误：

```
XY xy;
xy.f();      //错误，不知道应该调用X::f还是Y::f。
```

因为在检查存取保护权限之前，C++先检查了语句的二义性，所以由于私有函数X::f的存在，使试图调用Y::f的操作变得具有二义性。

因此，理想情况下，程序库应该对类的所有私有成员都应用命名约定。然而，这样做却会给程序库的开发者增加很多负担。由于在实际中这类冲突的出现频率是很少的，因此大多数程序库都不会对私有成员名称应用命名约定。

6.1.6 namespace（名字空间）结构

为了减少需要加上前缀的全局名称数量，程序库应该使用名字空间结构。例如：

```
namespace Companyxyz_libabc {
    class Widget {
        //...
    };
    void f();
    //...
};
```

由于所有的一切都被内嵌在名字空间里面，所以Widget和f都不需要前缀了。因为名字空间是在全局作用域内的，所以它也要符合上一小节所述的命名约定。在每个典型程序库中（即使是一个很大的程序库），名字空间的数目都会很少，因此那些很费解的名称数目相应也就会变得很少了（因为位于不同的名字空间内部）。

对于在名字空间Companyxyz_libabc内声明的名称，下面是程序员使用它们的方法：

```
Companyxyz_libabc::Widget widget;
Companyxyz_libabc::f();
```

如果用户愿意，他们也可以不书写名字空间的名称，依靠using结构语句来实现：

```
using namespace Companyxyz_libabc;
Widget widget;
f();
```

使用名字空间的唯一缺点就是：它是一个新加的C++特性；只有等到所有的编译器都实

现了名字空间，我们的代码才具有很好的移植性。

6.2 宏名称

在C++中，预处理宏的作用域和其他名称的作用域是不同的。某个宏名称的使用可能会和其他的宏名称（或者标识符）发生冲突。幸运的是，几乎所有宏的使用都是可以从C++中消除的。

6.2.1 宏名称冲突

考虑程序库LIB-A，并假设LIB-A在它的一个头文件中定义了下面的宏：

```
libA.h:
    #define MACRO stuff
```

假设LiB-B也在它的一个头文件中定义了一个MACRO宏：

```
libB.h:
    #define MACRO different stuff
```

那么，同时使用这两个程序库的程序将会出现一个“重新定义”错误：

```
user.c:
#include<libA.h>
#include<libB.h>          // 错误，宏MACRO发生了重新定义。
//...
```

即使除了LIB-A外，没有其他程序库和用户代码命名宏MACRO，LIB-A中宏的存在仍然有可能会导致冲突。假设LIB-B提供了下面的类：

```
libB.h:
    class X {
        void MACRO();
        //...
    };
```

那么对那些在包含libB.h之前，包含libA.h的代码，将会改变X成员函数的名字，这可能会导致非预期的结果。

宏还有其他比名称冲突更加严重的缺点。首先，在大多数系统上，宏名称在调试期间并不是可用的；其次，宏并不是类型安全的。考虑下面的宏：

```
#define ABS(X) ((X) > 0 ? (X) : -(X))
```

宏的作者希望只对整数或者浮点数调用上面这个宏，但是，下面的调用可以顺利通过编译，并不存在任何出错消息或警告：

```
ABS('q');          //并没有出现任何编译器诊断信息。
```

为了减少由宏引起的问题到最少，程序库应该遵照下面的两条规则：

1. 尽可能地少定义公共宏（公共宏是指在公共头文件内定义的宏）；
2. 对那些不能去掉的公共宏，使用命名约定。

6.2.2 去掉宏

C++提供了许多用于去掉宏的机制。事实上，在现实的C++代码中，所有预处理程序的用途（除了#include）都属于下面类别的其中一种：

- 注释掉代码，例如：

```
#if 0
int not_needed_anymore;
#endif
```

- 符号常量，例如：

```
#define TABLESIZE 1024
```

- 内联函数，例如：

```
#define CLEAR(X) ((X) = 0)
```

- 泛型函数，例如宏

```
#define ABS(X) ((X) > 0? (X) : -(X))
```

可以对整数和浮点数调用：

```
ABS(-7);
ABS(-7.0);
```

- 泛型类型，例如：

```
#define Stack(T) Stack__ ##T
#define Stackdeclare(T) class Stack(T) { /*...*/}
Stackdeclare(int);
Stackdeclare(char);
//...
Stack(int) s1;
Stack(char) s2;
```

- 语法扩展。例如，假设Set是一个描述集合的类，那么程序员可以定义一个FORALL宏，并且用这个宏编写下面的代码：

```
Set<int> s;
int i;
FORALL(i, s)
//...
```

- 声明宏，例如：

```
#define DECLARE_MEMBERS(X) X(); ~X(); X(const X&)
//...
class Widget {
public:
    DECLARE_MEMBERS(Widget);
    //...
};
```

- 代码版本管理。例如，下面的函数在调试版本和非调试版本中将执行不同的代码：

```
void f() {
#ifdef DEBUG
```

```
    //调试版本。
#else
    //非调试版本。
#endif
}
```

在C++中，除了声明宏和代码版本管理外，其他所有预处理程序的使用，我们都可以很好地进行替代。譬如，注释预处理程序可以用下面的注释来替代：

```
//int not_needed_anymore;
/* int also_not_needed_anymore; */
```

(上面注释的一个很小的缺点就是：对于很大的代码段，把这些代码段全都包含在#if 0 ...#endif中要比在每一行的开头都使用 // 简单些。)

符号常量预处理程序可以使用const关键字或者enum值来代替：

```
const int TABLESIZE = 1024;
enum { TABLESIZE = 1024 };      //这个也可以
```

一个非泛型内联函数的预处理程序可以使用真正的内联函数来代替。例如，如果前面给出的CLEAR宏只是接收一个int类型的参数，那么我们可以使用下面的内联函数来代替这个CLEAR宏：

```
inline void clear(int& x) { x = 0; }
```

有时，代替使用某个内联函数预处理程序，需要某些额外相关程序代码的改变，考虑下面的代码：

```
#define CONTROL(c) ((c) - 64)
//...
switch(c) {
case CONTROL('a'):  //...
case CONTROL('b'):  //...
//...
}
```

因为对于C++而言，用case标签作为函数调用是非法的，所以只用内联函数来代替CONTROL宏将会导致非法的代码。然而，我们可以如下转换代码：

```
inline char decontrol(char c) {
    return c + 64;
}
//...
switch(decontrol(c)) {
case 'a':       //...
case 'b':       //...
//...
}
```

(上面的代码由于把加法运算放在了运行时，那么这个版本的switch将比使用宏的switch慢一些。)

实现泛型函数（或者类型）的预处理程序可以用函数模板或类模板来代替：

```
template<class T>
T abs(const T& t) { return t >0 ? t : -t; }
Template<class T>
Class Stack { /*...*/ };
```

最后，C++语法扩展几乎可以被一个或者多个C++类所代替。例如，与其定义FORALL宏，C++程序员更愿意定义一个Set_iter类，并且可能编写出下面代码：

```
Set<int> s;
int i;
Set_iter<int> iter(s);
while(iter.next(i))
    //...
```

与使用宏FORALL的代码相比，上面的代码只是在简洁性上差了一点。

6.2.3 宏的命名约定

如6.1.5小节所述，对那些不能替换的公共宏，应该对它们实行命名约定。例如：

```
#define COMPANYXYZ_LIBABC_MACRO stuff
```

对于所有的公共宏，一定要牢记遵守命名约定。特别地，在公共头文件中，对于那些用于预防多重包含的宏（每个程序库的头文件都应该预防这种情况），更应该遵守命名约定：

libA.h:

```
#ifndef COMPANYXYZ_LIBABC_WIDGET_H
#define COMPANYXYZ_LIBABC_WIDGET_H
//...
#endif
```

而在程序库实现内部定义的宏则不需要遵守命名约定：

libimpl.c

```
#define MACRO...
```

这个宏就不会和其他程序库（或者用户代码）发生冲突[1]，而且，libimpl.c的实现者大概也知道这个宏不会和翻译单元内其他的代码发生冲突。

6.3 环境名称

对于6.1节和6.2节讨论的名称，所有都是出现在C++程序库源代码里面的名称。但是，C++程序库使用的名称并不都在源代码里面。譬如，一个典型的程序库提供了几个头文件，每个头文件都有一个名称。而且，程序库可能还一起提供在线助手册，或许还提供了某些可执行文件，或许还使用了某些环境变量，所有这些事物也都有名称，我们把这些名称都统称为环境名称。

不同程序库的环境名称也会发生冲突。例如，假设两个程序库LIB-A和LIB-B都提供了一个公共头文件Widget.h。如果LIB-A和LIB-B的用户把所有程序库的头文件都放在同一个

[1] 译注：请注意，上面是实现文件.c，而不是头文件.h。

目录里面，那么在链接头文件的过程中，后链接头文件的 Widget 类将会把先链接头文件的 Widget 类覆盖，而这种覆盖显然不是我们所期望的。

为了避免冲突，假设用户把不同程序库的头文件分别放在不同的目录下，并且在提供给编译器的#inlcude 路径中，指定这些目录；遗憾的是，这个方法仅仅是转移了冲突，而并没有真正解决冲突，如果用户编写了代码

```
#include<Widget.h>
```

那么预处理程序只会使用在#include 路径里首先找到的 Widget.h，而这不一定是我们所期望的 Widget.h。所以说，重新排列#include 路径（即把头文件放在不同的目录下）只是把问题转移给了别的头文件。

为了减少环境名称冲突的可能性，所有和程序库相关的文件都应该在一个单独的目录下，而且这个目录的名称是唯一的。例如，公司 XYZ 的所有 ABC 库文件都应该在名为 Companyxyz_libabc 的目录下。另外，所有引用程序库文件的代码都应该使用和程序库唯一目录相关的路径名称。例如，如果 Widget.h 是在 Companyxyz_libabc 目录的 incl 子目录下，那么程序员应该如下#include Widget.h 头文件：

```
#include<Companyxyz_libabc/incl/Widget.h>    //正确的方法。
```

而不是这样：

```
#include<Widget.h>                                    //不好的方法。
```

程序库的用户和实现者都应该遵从这种约定。如果实现者没有遵从这种约定，那么当程序库源代码用户在他们的环境中编译源代码的时候，将会发生冲突。遗憾的是，我们将在 9.9 节看到，减少环境名称冲突将会制约程序库的可移植性。

在某些系统上面，文件名称是不区分大小写的；因此，例如，名为 Widget.h 的头文件可能会和名为 widget.h 的系统文件发生冲突。

6.4 Unclean 程序库

如果每个全局名称、公共宏名称和环境名称都遵守 6.1 节到 6.3 节描述的命名约定，那么我们就称这个程序库是名称 clean 的，或简称 clean。有时，unclean 的程序库也不会有导致冲突的可能。譬如，我们考虑某个名为 UNCLEAN 的 unclean 程序库，并假设对于会和 UNCLEAN 出现在同一个程序里的每一部分 C 代码，下面条件中至少其中一个总是成立的：

1. C 是 clean 的。

2. 在编写 C 代码的时候，我们已经知道 C 代码将会和 UNCLEAN 出现在同一个程序里面。

那么，可以很容易地看到，UNCLEAN 将不会导致冲突。

什么样的程序库才既是 unclean 又是安全的呢？标准 C++程序库就是一个很好的例子：在编写每部分将会出现在标准 C++程序库里面的代码之前，我们都会慎重考虑 unclean 的可能性。另一个例子是只为作者个人使用的 C++程序库，程序库中的代码就只会和该作者编写

的其他代码一起使用，因此这些都符合上面的条件 2，或者就是那些来自其他程序库并且满足——或者应该满足——条件 1 的代码，由这些代码组成的程序库也是符合条件的。当然，如果某个程序库只是针对个人是 unclean 的，那么对于那些希望使用这个程序库的其他人（可能是这个人的同事），将可能会考虑到名称冲突，而未能使用这个程序库。我们还应该注意：对于那些刚开始只作为个人用途的程序库，到最后通常并不局限于个人使用——特别是那些有用的程序库更是如此。

6.5 Good-Citizen 程序库

假设程序库 LIB-Q 希望从一个特殊的内存池里分配它所有的空闲存储区对象，那么，它可以自己定义一个全局操作符 new。然而，这个方法也是有缺点的：现在所有调用 new 的操作——而不仅仅是指 LIB-Q 内部调用这个操作——都将会调用 LIB-Q 自己定义的这个操作符 new。也就是说，如果用户想要链接某个其他的程序库，这个程序库所有的 new 操作也同样会调用 LIB-Q 的 new，这就和其他程序库所期望的行为发生冲突。

因此，我们称上面的 LIB-Q 并不是一个 good-citizen 程序库。它霸占了全局资源（操作符 new）的所有权。由于占用了某个全局资源，这个程序库就会和其他希望占有这个资源的程序库（或者用户代码）发生冲突。而一个 good-citizen 程序库则会避免霸占任何全局资源。C++的另一些全局资源包括：全局操作符 delete，new 处理程序，terminate 函数和用于异常处理的 unexpected 函数。不仅如此，good-citizen 程序库也不会霸占任何特定于应用程序的资源。例如，在一个窗口系统应用程序中，good-citizen 程序库不会霸占主窗口的所有权。

有时，我们会有意地设计一个非 good-citizen 的、非个人使用的程序库。例如，由于某种原因，在用户程序里面，对于任何用户在任何位置创建的空闲存储区对象，我们的程序库都需要把这些对象分配在特殊的内存池里面。于是，如果一定要得到这种行为的话，我们可以定义自己的 new 函数，并要求我们的用户使用这个 new 函数，从而达到目的。对于这样的设计，虽然最终的程序库并不是一个 good-citizen 程序库，但如果我们不选择这种设计方式，而选用其他的设计方式将会给我们的用户带来更大的问题，所以也就不得不选择这种设计方式了。

6.6 总结

当两个程序库发生冲突的时候，要在同一个程序里同时使用这两个程序库将会是困难的，或者是不可能的。为了最大化重用性，程序库设计者应该避免和其他的代码发生冲突。对于程序库定义的所有的全局名称、公共宏和环境名称，命名约定和命名空间结构的使用都是必不可少的，除非所给程序库可以既是 unclean 的，又是安全的。Good-citizen 程序库避免了另外一种形式的冲突，即由于霸占全局资源或者应用程序的特定资源而导致的冲突。

6.7 练习

6.1 (**) 有些C++规则规定了什么样的声明和定义可以合法地出现在程序的全局作用域里面，而不导致冲突。准确地说出这些C++规则。

6.2 解释C++程序库为什么不能在全局作用域内定义一个名为Root或者Object的类。

6.3 假设你在设计一个用于房地产买卖的程序库，第一次设计的类的声明如下：

```
class Listing {
public:
        Listing();
        void reduce_asking_price(Price new_price);
        //...
private:
        Price asking_price;
        bool has_attached_garage;
        int nbedroom;
        //...
};
```

a. 使用一个合适的命名约定来修正上面的声明，从而使你的用户之间不会在类的名称上发生冲突。

b. 为了方便你的用户使用修改后的Listing类，编写这个类的说明文件。

c. 使用命名空间结构，修改Listing类的声明。

6.4 在C++包含命名空间结构以前，某些程序库设计者使用一些作用域（scoping）类来减少名称冲突。作用域类是指用于提供作用域的类，而不是用于创建对象的类。例如，与其如下声明：

```
int abc_val;
void abc_func();
class ABC_Widget { /* ... */ };
```

还不如这样声明：

```
class ABC_lib {
public:
    static int val;
    static void func();
    class Widget { /* ... */ };
private:
    ABC_lib();
};
```

因此，上面的标识符val、func和Widget都会在ABC_lib的作用域里面；以后就可以使用作用域操作符来引用这些标识符，如下所示：

```
int i = ABC_lib::val;
ABC_lib::func();
```

```
ABC::Widget w;
```

a. 我们为何要为类 ABC_lib 提供一个私有构造函数呢？

b. 对于在作用域类内部声明的宏，它们的作用域是什么？例如，下面的 some_macro 的作用域是什么？

```
class ABC_lib {
        #define some_macro x
        //...
};
```

c. 如果我们希望在作用域类里面内嵌一个模板类，结果将如何？示例代码如下：

```
class ABC_lib {
        template<class T>
        class ABC_Another_Widget { /*...*/ };
        //...
};
```

d. 如果程序库使用了一个作用域类（或者一些作用域类），那么是否可以在全局作用域避免由于名称而导致的冲突。给出你的答案。

6.5 假设程序库 LIB-A 中的头文件和程序库 LIB-B 中的某个头文件使用了同一个宏名称，那么对于同时使用这两个程序库的程序，如何才会引起冲突呢？C++编译系统能否检测到这种冲突？给出你的解释。

6.6 如果在程序里面内嵌一个如下的#ifndef 结构，那么我们就可以监视宏的定义：

```
#ifndef MACRO
#define MACRO ...
#endif
```

但是，即使所有的程序库（或者用户代码）都如上监视它们所有的宏定义，我们还是不能消除 6.2 节所描述的冲突。为什么？

6.8 参考文献和相关资料

C++规则——指定什么样的声明和定义才可以合法地出现在程序的全局作用域中——正处于 C++ ANSI/ISO 委员会的积极讨论之中。（特别地，委员会还必须澄清“一处定义原则”。另外，在什么样的条件下，类的定义才可以出现在别的翻译单元中。如果两个类的定义是相同的，那么这意味着什么呢？）当这本书出版的时候，这些规则都可以在 ANSI/ISO 的工作底稿[ANS94]中找到

Kendall 和 Allin 在[KA94]中进一步解释了类链接的概念。

Berlin 在[Ber90]讨论了有关 C++程序库之间冲突的内容。

第7章

兼容性

我们的任务不在于质疑为什么，而在于编码和认清那些需要。

——Rob Murray

可重用代码的大多数提供者都需要关心代码版本之间的兼容性。代码版本可以提供各种不同形式的兼容性：源代码兼容性、链接兼容性、运行兼容性和进程兼容性。

在这一章里，我们将详细说明这些各种不同形式的兼容性，并讨论如何提供这些兼容性。我们还建议为非兼容性特性提供必要的文档；而对于用户那些依赖于可重用代码中无文档化特性的可能性，我们提醒程序库设计者要充分重视这种可能性。

7.1 向后和向前兼容性

一个程序库如果不断升级，那么使用这个程序库的用户通常都会升级到程序库的最新版本。例如，用户可能需要开发新的功能、优化程序、排除程序错误，或者和其他使用最新版本程序库的代码保持同步；显然，使用最新版本的程序库通常都有助于实现这些目的。如果从程序库的版本 m 升级到版本 n 的操作是容易的（指升级下一节所定义的各个因素都是容易的），那么我们就说：对版本 n 而言，版本 m 具有向后兼容性；对版本 m 而言，版本 n 具有向前兼容性[1]。

向前兼容性和向后兼容性犹如一枚硬币的两面：如果已经提供一个很好的向前兼容性，那么提供向后兼容性就会容易很多。于是，在这一章剩下的内容里，我们将只从为程序库用户提供向后兼容性的角度来讨论兼容性。

[1] 并非所有的人都如我们定义的那样使用术语向后和向前兼容性。我们的定义只是最普遍认可的用法。

7.2 兼容性的形式

先考虑C++程序员创建一个程序的具体过程，图7.1给出了这整个过程。首先，程序员编写能够实现某种需要行为的源代码；接着，源代码被编译成目标代码，然后链接一个或者多个翻译单元的目标代码，并且创建可执行代码；最后，执行可执行代码，在底层机器创建了一个进程。如果上面的每一步都可以准确无误地实现，那么最后的程序将可以表现所期望的行为。

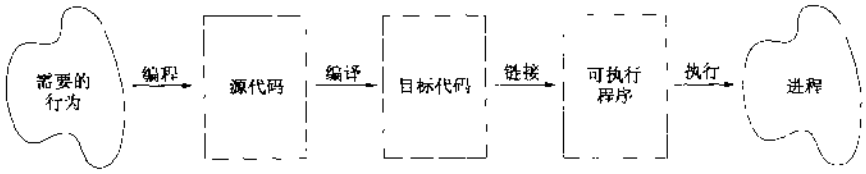


图7.1 程序的创建过程

对于使用m版本程序库编写、创建、执行的程序，如果需要升级到n版本的程序库，那么，升级的难易程度要取决于程序创建过程中需要修改的步骤：

- 如果没有用户需要修改他们的源代码，那么程序库的版本n和版本m就具有源代码兼容性。
- 如果没有用户需要重新编译任何代码，那么版本n和版本m就具有链接兼容性。
- 如果没有任何目标文件等需要重新链接，那么版本n和版本m就具有运行兼容性。
- 如果使用m版本并且正在执行的程序可以立刻升级到n版本，那么版本n和版本m就具有进程兼容性。

让我们更深入地考虑源代码兼容性的定义："没有用户需要修改他们的源代码"这句话究竟是什么意思呢？有时，程序库的改变会迫使程序库用户修改他们的源代码，其潜在原因并不是程序库开发者所能预见到的。例如，假设我们改变了非内联函数f的实现，而并没有改变f的语义；并且我们并不知道我们的一个用户正处于资源紧张状况，如果我们的改变导致了这个用户的程序超出了该程序的代码大小限制，那么用户就不得不采取某种方式改变程序源代码，来符合代码大小的限制。虽然如此，但我们通常都不会认为这样的改变是不兼容的（即认为这样的改变应该是源代码兼容的，但需要修改定义）。

因此，我们需要重新定义源代码兼容性的。因此，对于程序库的某个改变，只有由于语义发生改变而修改用户代码，而不是由于别的原因而修改用户代码，我们才认为这个改变是源代码不兼容的：

对于每个合法的程序，如果使用m版本的程序库和使用n版本的程序库都是合法且语义相同的，那么我们就认为程序库的n版本和m版本是源代码兼容的。

根据这个定义，我们对 f 函数的修改就是源代码兼容的。

这个定义虽然比原先的定义更加准确，但它也引发了一个问题：我们所说的“每一个程序”里的“每一个”意味着什么呢？是否指每个可能的程序，或者每个已有的用户程序？我们把前者称作理论兼容性，而把后者称作实际兼容性。自此以后，我们所指的“源代码兼容性”（如果没有限定词的话）都是指“实际源代码兼容性”。对于其他的兼容性形式，理论兼容性和实际兼容性也是有类似的区别。在 7.4 节和 7.7 节，我们将更加详细地讨论兼容性的形式。

7.3　理论源代码兼容性

理论上，C++程序库的每个改变几乎都具有源代码不兼容性。我们相信：对于程序库的每个重大改变，我们都可以构造一个假定的用户程序，这个用户程序为了适应程序库的改变，而不得不修改源代码。例如，考虑一个看似无关紧要的程序库改变：给类增加一个成员函数。下面是包含已增加函数的类 X：

```
class X {
public:
    void added();
    //...
};
```

现在考虑下面假定的用户代码：

```
class User_x : public X {
public:
    void f();
    //...
};
```

假设用户程序包含了一个全局作用域函数的定义，这个函数的名称恰巧也是 added，而且 User::f 函数的实现如下：

```
extern void added();
void User_x::f() {
    added();
    //...
}
```

使用没有 added 函数的 X 版本，User::f 函数内部将调用全局作用域的 added；而使用带有 added 函数的 X 版本，User::f 函数调用的是 X::added[1]。于是，我们就必须改变用户的代码，因为调用语义已经发生改变了。因此，从理论上讲，给类增加一个公共成员是源代码不兼容的。

“Okay”，你可能会说，“但这是对程序库接口的改变。我认为对程序库实现的改变会是

[1] 译注：VC6 测试的结果是确如此。

理论源代码兼容的，对吗？”很遗憾，是不对的。假设我们希望添加的成员函数是一个 X 的私有成员：

```
class X {
private:
    void added();
    //...
};
```

因为在 C++中，存取保护的检查是发生在名字查找之后的，所以当我们添加了这样的改变之后，这个和前面相同的、假定的用户程序将会变成非法程序[1]。因此，从理论上讲，即使这样的改变也是源代码不兼容的。

7.4 实际源代码兼容性

大多数程序库编写者都不能检验每个用户的程序，因此，对于程序库编写者而言，如果某个给定的程序库发生了改变，那么想要决定这个改变是否会影响实际兼容性是很困难的。然而，某些对这些改变的质疑，经常可以给我们提供答案。例如，类 X 并不是为了派生而设计的类；虽然某个顽固的用户会从 X 派生，但很可能没有用户会这么做（看 4.3 节的讨论就可以知道：从一个不是设计用来派生的类中派生的难度）。如果我们假设没有用户会从 X 派生，那么我们对类 X 的修改就具有实际源代码兼容性了。

通常，那些既没有删除也没有改变已有功能的更改，在实际中都是源代码兼容的。例如，下面的更改就是实际源代码兼容的：

- 增加一个成员函数；
- 授权友元关系；
- 放宽成员或者基类的保护级别（就是说，把 private 改变成为 protected 或者 public，或者把 protected 改变成为 public）；
- 把先前的外联（out-of-line）函数改变成内联函数；
- 把内联函数改变成外联函数。

另一方面，删除功能的改变往往都是源代码不兼容的。例如，假设我们稍后想要从我们的程序库中删除 X::added 函数，那么每个调用这个函数的程序将不得不进行修改。当然，如果我们知道已经没有用户使用 X::added 函数，那么就可以安全地删除这个函数，从而没有破坏兼容性。

借助于被称为弃用（deprecation）（或者除黑）的移植技术，用户可以更容易地处理源代码不兼容性。与其把函数从程序库中删除，不如在相应文档中建议不再使用这个函数。于是，过去使用这个函数的程序可以继续运行，新编写的程序将不再使用这个函数（我们希望是这样）。程序库用户也就可以等到方便的时候才修改他们的代码，从而来适应程序库的变化。等

[1] 译注：VC6 测试结果的确如此。

到弃用的功能真正从程序库中删除的时候（如果有这个必要的话），那些使用这个功能的程序将不能再继续运行，也就没有改变源代码的必要了。

某个函数即使被弃用了，但它的描述还应该依然在文档中存在，因为有些程序员还需要维护使用这个函数的旧程序代码。而且，当我们弃用一个函数的时候，我们可以用手册的特定细节来文档化这个函数，并把剩余文档中指向这个函数的引用都删除。如果最终希望从程序库中删除这个函数，我们还必须通知用户这个删除操作所带来的影响。

弃用技术不仅仅可以用于单个函数，它还具有其他层次范围的应用。我们可以弃用的对象有：函数组、类、类组，甚至整个程序库。

7.5 链接兼容性

对于已经拥有很多现有软件（这里软件概念指的是某个项目的一部分）的程序员，可能不会为了把他们所使用程序库升级到新的版本，而重新编译所有的代码。因为对于某些项目而言，重新编译可能需要花费好几天的时间。而且在某些情况下（很少），项目并不具有所有源代码的存取权限，而目标代码则是创建自这些源代码的；另外，在项目的生成过程中，有些细节是必须要依赖于软件的，而软件在这时可能已经不再是可获得的。因此，许多用户更希望新版本的程序库可以和旧版本的程序库具有很好的链接兼容性；那么，一个程序希望从版本 m 升级到具有链接兼容性的版本 n，程序员需要做的事情只是进行重新链接。

遗憾的是，提供链接兼容性（有的也称为“二元兼容性”）实现起要来比表述起来困难许多。随着程序库的发展，程序员对程序库所作的大多数改变都会导致和旧版本的不兼容。在这一节里，我们将讨论哪种改变是链接兼容的，而哪种改变不是。在 8.2 节中，我们还给出了如何设计类和继承体系，来提高程序库链接兼容性改变的可能性。

我们平时经常看到的链接兼容性改变是指：改变一个外联函数的实现，而并不用改变函数的接口。于是，任何没有影响程序库公共头文件的改变都必定是链接兼容的。另外，这些改变是有限制的，但这些限制包括：对用户不能构造的类型的改变、对用户不能存取的对象的改变、对那些实现细节不会出现在用户目标代码中的改变（通过内联实现）。在这一节的剩余内容里，我们将讨论那些影响程序库头文件的改变。

任何源代码不兼容的改变同时也是链接不兼容的。例如，下列的改变就既是源代码不兼容，又是链接不兼容的：

- 删除一个函数（从程序库接口中删除）。
- 改变一个函数的返回类型或者参数类型。
- 提高成员的存取级别（即把 public 改变成 protected 或者 private，或者把 protected 改变成 private）。

下面是另外的一些链接不兼容改变：

- 任何对类的设计布局或者大小的改变——包括添加、删除数据成员或者改变类的大

小、偏移量等。

- 把一个外联函数改变成为内联函数。因为编译使用旧版本的代码需要外联函数的定义；然而在新版本中，这样的定义已经不再存在了。
- 几乎所有对内联函数的改变。
- 给以前不具有虚函数的类增加一个虚函数。任何具有一个或者多个虚函数的类都会拥有一个虚函数表，在大多数C++实现中，这个类型的任何对象都会包含一个指向虚函数表的指针。

然而，某些对程序库头文件的重大改变也可以是链接兼容性的，下面是某些链接兼容性的改变：

- 增加一个非虚函数——内联或者外联，这个函数提供的功能既没有重载也没有隐藏已存在函数的功能。
- 放宽某类的保护级别（把基类从private类型改变为protected或public，或者从protected类型改变为public。
- 授予友元关系。

把一个内联函数改变成为外联函数可能是（也可能不是）链接兼容的。如果用户拥有函数旧版本的目标文件，并且链接新版本的程序库，那么用户可能不会调用函数的新版本。因此，如果在新版本中函数的语义发生了改变，那么这个改变就是链接不兼容的。另一方面，如果新版本的函数和旧版本的函数具有相同的语义（例如，新版本只是具有更高的效率），并且原来的用户代码实现可以使用新版本的程序库，那么这个改变就是链接兼容的。

在实际中，许多改变的链接兼容性要取决于用户使用的编译系统。例如，考虑下面的改变：

- 放宽成员变量的保护级别。不同的实现可能会选择不同的存取标识符来排列成员变量。例如，C++实现可能会以变量声明的先后次序来排列成员变量，由于放宽成员变量的保护级别不会改变这个排列次序，所以这个改变是链接兼容的。如果C++实现是利用存取权限分组来排列成员变量，那么放宽成员变量的存取权限将会导致类对象的成员结构发生变化，因此也就是链接不兼容的。
- 在类的定义中重新排列虚函数的声明。某些C++的实现依据虚函数声明的顺序，在类的虚函数表中放置每个函数的入口，基于这样的C++实现，重新排列虚函数的声明将会是链接不兼容的。而对其他的实现，如果虚函数表的顺序和虚函数的声明顺序无关——例如，只是照函数名称的字母顺序排列；那么，基于这样的实现，重新排列虚函数的声明就是链接兼容的。
- 给已经含有一个或者多个虚函数的类再添加一个新的虚函数。很明显，这样的更改将会改变虚函数表的大小和布局。然而，如果这个被使用的C++实现是依照函数声明的前后顺序来布局虚函数表的，并且这个新加的虚函数声明位于所有其他虚函数之后，那么对于已经存在的使用虚函数表的代码，新虚函数的增加将不会影响这些代码。
- 给函数增加一个异常规范，更改函数的异常规范，或者删除函数的异常规范。对于大

多数异常处理的实现，上面这些改变都是链接兼容的。

当然，我们并不提倡依赖于用户编译系统的实现。如果必须依赖于这样的实现，一定要确认编译系统使用什么来编译你的代码，而且无论你依赖的是编译系统上面的哪一方面，都必须彻底弄清楚编译系统是如何实现这些方面的。

7.6　运行兼容性

如果程序库的版本 n 和早期的版本 m 是运行兼容的，那么只要通过再次执行程序，使用版本 m 的程序就可以得到升级。在这里，为了提供运行兼容性而通常使用的机制就是动态链接机制

假设我们创建了一个程序，这个程序非动态（即静态）地链接两个对象（t1.obj 与 t2.obj）和一个档案文件 my.lib:

```
% link t1.obj t2.obj my.lib
```

如果这个链接成功了，那么生成的可执行文件将包含一份含有 t1.obj 和 t2.obj 的代码、以及 my.lib 中的目标文件的拷贝，这份拷贝包含有程序使用的所有函数或者数据成员的定义。

现在假设进行动态链接。虽然对于不同的系统，动态链接的机制可能会有所差异，但程序员一般都是先创建（或者还有可用的）动态链接库（DLL）——也就是说，适用于动态链接的档案文件。（在某些系统中，DLL 也被称为共享程序库。）假设我们已经创建了一个名为 my.dynlib 的 DLL，它包含 my.lib 中的所有代码。如果我们使用 my.dynlib 来进行链接：

```
% link t1.obj t2.obj my.dynlib
```

那么生成的可执行文件将包含一份含有 t1.obj 和 t2.obj 的代码、但不包括 my.dynlib 中的代码的拷贝。动态链接程序的运行时支持将会促使调用定义在 my.dynlib 内部的函数以执行适当的代码。

动态链接的许多实现也提供了 DLL 的版本管理机制。基于这样的实现，当 DLL 被创建的时候，这个 DLL 就已经被赋予了一个版本号。这个版本号将会自动地赋给我们的程序，或者我们可以亲自显式地指派版本号。我们还可以指定（使用某些依赖于系统的机制来实现）这个版本号和 DLL 的哪些其他的版本号是链接兼容的。假设我们使用 my.dynlib 的 m 版本来链接我们的程序；在运行时，对定义在 my.dynlib 中函数 f 的调用，将会执行位于 my.dynlib 可用的、最高版本的 my.dynlib 中的 f 函数代码；这里的最高版本指的是和版本 m 链接兼容的最高版本。

使用动态链接来使程序库的版本 n 和版本 m 具有运行兼容性，通常都会包括下列步骤：

1. 确认版本 n 和版本 m 是链接兼容的。
2. 给版本 m 和版本 n 分别创建 DLL。
3. 标明（指出）版本 n 是和版本 m 链接兼容的。

第二步可能是比较困难的。某些 C++实现，当和某些动态链接实现结合使用的时候，不

能正确地运行。考虑下面的代码：

```
class X {
public:
    X();
    //...
};
X x1;
Void f() {
    Static X x2;
    //...
}
```

当C++实现和动态链接互相结合的时候，如果这个实现代码是位于DLL里面的，那么对象x1或x2（或者两个都）是永远也得不到初始化的。

注意，如果程序库遵从将在12.1节讨论的建议，那么我们将不会如上声明x1。如果动态链接的实现可以正确初始化局部静态变量（如x2），那么在实际中，给定的C++实现和动态链接机制互相结合，并一起使程序正确运行的可能性还是很高的。

7.7 进程兼容性

如果程序库的版本n和早期的版本m是进程兼容的，那么对于用版本m编写和创建的程序，当它们在执行的时候，就可以进行升级。例如，我们要在飞机飞行的时候升级它的航空控制系统，并要求飞机整个升级过程都保持飞行状态。然而，进程兼容性是很难实现的，我们也很少看到这种兼容性。另外，进程兼容性使用的机制是高度依赖于系统的，并且也超出了本书研究的范围。这里，我们只是给出任何进程兼容性系统所需要达到的几个要求。

假设有一个只包含类X的程序库，并且程序库的版本m和版本n是链接兼容的。如果我们对正在执行的程序从版本m升级到版本n，那么下面两点（至少两点）是肯定会发生的：

- 在程序执行并调用版本n的过程中，对类X的所有成员函数调用都必须进行升级。
- X对象的所有现存实例都必须更换成使用n版本的X对象的实例。

其中，对X的现存实例升级的一个办法就是：在做任何事情之前，检测X的每个成员函数，来判断这个被调用的X对象是否已经被升级了。如果没有，就升级这个对象：

```
//程序库的n版本。
void X::f() {
    if(!is_version_n(this)) {
        //升级*this到版本n。
        //...
    }
    //...
}
```

对于那些对如何提供进程兼容性细节感兴趣的程序员，应该参考Coplien的书[Cop92]。

7.8　文档化不兼容性

程序库的每个版本都应该用文档记录这个版本和以前版本的源代码不兼容性和链接不兼容性，而且，那些真正为用户考虑的程序库设计者，还应该对每个不兼容性给出解释，从而指导用户如何更改用户代码来升级程序库。文档的内容举例如下：

函数 Widget::operator++已经被删除了，现在可以使用 Widget::next 来代替它。

所有的兼容性注释都应该是程序库文档的一部分。

并不是程序库的所有用户都会把他们的程序升级到新版本。例如，某些用户可能不会选择升级到*.0 版本（例如 1.0、2.0、3.0 等版本），因为这些版本总是被认为含有很多错误。于是，某个用户可能要离 3.0 的最近的版本升级到 3.1 版本（跳过 3.0 版本）。让我们假设离 3.0 最近的版本是 2.9 版本。那么从 2.9 版本升级到 3.1 版本，用户不但要遵循从 2.9 到 3.0 的升级指导，而且还要遵循从 3.0 到 3.1 的升级指导。除了非常麻烦之外，这整个遵循过程的开销还是非常昂贵的——即使用户不需要使用 3.0 版本，但由于要遵循升级规范，用户就不得不购买 3.0 的说明文档。因此，为了程序库用户的方便，程序库提供者应该同时提供本版本和前面几个版本之间不兼容性的升级指导文档（10.2.4 小节进一步讨论兼容性和文档化之间的关系）

7.9　非文档化特性

假设我们的程序库提供了一个类 Gizmo，这个类含有一个 operator<<：

```
ostream& operator<<(ostream& o, const Gizmo& g);
```

并假设我们如下文档化这个操作：

把 g 的可读表示输出到流 o 中，并返回 o。

上面这个文档既没有描述生成的输出格式，也没有保证输出格式在将来发布的新版本中不会发生改变。因此，我们可以认为改变 operator++的输出格式可能不会破坏源代码的兼容性。遗憾的是，如果 Gizmo 是一个得到广泛使用的类，那么其他地方的某些用户可能会编写依赖于输出格式的代码；如果我们改变了原来的格式，那么为了处理这种新格式，这些用户就不得不对他们的源代码进行更改。因此，某些事物没有记录在具体文档中并不意味着，对这个事物进行改变不会破坏兼容性。

用户之所以依赖于程序库非文档化的特性大多是基于下面两种原因。第一，用户没有别的选择（即必须依赖于非文档化的特性）。例如，Gizmo 类的用户需要编写一个程序 P1，它主要是用于分析另一个程序 P2 的输出；如果 P2 调用了 Gizmo 的流插入操作符（即上面的 operator<<操作符），那么 P1 就必须依赖于这个操作符（即 Gizmo 流插入操作符）的输出格式。遗憾的是，程序库设计者不太可能避免这种由非文档化特性带来的依赖性。

用户依赖于非文档化特性的第二个原因是：用户可能热衷于读取头文件。例如，假设类

Gizmo 的实现包含了一个成员函数 useful_function，并且我们需要在我们程序库的其他类中调用这个函数。其中一个实现方法就是，可以把 useful_function 声明为 private 类型函数，并把其他的类声明为 Gizmo 的友元类：

```
class Gizmo {
    friend class A;
    friend class B;
    friend class C;
    friend class D;
    friend class E;
    friend class F;
    friend class G;
    //和其他的所有友元类
    //...
    private:
        void useful_function();
};
```

然而，当我们把一个类声明为 Gizmo 的友元类时，我们给这个类的权限不仅仅是访问 useful_function，而是整个 Gizmo 的实现。为这么多类都提供 Gizmo 的整个实现往往是很容易产生错误的。一个更好的方法就是把 useful_function 函数声明为 public 类型，但没有给这个函数提供文档：

```
class Gizmo {
public:
    void useful_function();      //非文档化.
    //...
};
```

遗憾的是，那些热衷于读取头文件来寻找感兴趣函数的用户，可能会在他们的代码中调用（不该被外部调用的）useful_function 函数。为了避免这种调用，在程序库的头文件中应该给出如下的一段警告：

```
class Gizmo {
public:
    //警告：之所以把 useful_function 函数声明为公共函数，只是为了使这
    //个函数的实现更加容易。这是一个非文档化函数，程序库用户不应该调用这个
    //函数；也不能保证这个函数会在程序库将来的版本中继续存在。
    void useful_function();
    //...
};
```

而那些不顾上面的建议，仍然调用这个函数的用户，也只能自己为调用的后果负责。

7.10 总结

程序库开发者应该着重考虑：为他们现在的用户提供向后兼容性，并且预料向前兼容性，

从而他们才可以在将来的版本中提供更好的向后兼容性。程序库应该尽可能地提供源代码兼容性、链接兼容性和运行兼容性。某些程序库还要尽量提供进程兼容性。提供兼容性需要充分考虑对程序库所做的改变。弃用功能取代了删除功能，它将用另一种方式来提供源代码兼容性，并且还允许用户可以在方便的时候才改变他们的代码。

对于程序库版本之间的不兼容性，程序库设计者应该对此提供清楚的说明文档，并且附加用户程序的升级指导。程序库提供者还应该充分意识到用户依赖于程序库非文档化特性的可能性究竟有多大。

7.11　练习

7.1　假设你在开发一个程序库，它包含练习 1.2 讨论的 Path 类；而且在你的程序库版本 1 中，Path 类并不是一个自动规范化(canonicalized)的类，需要提供一个公共函数 canonicalize：

```
class Path {
public:
    Path(const char*);
    void canonicalize();
    //...
};
```

假设 Path 类的版本 1 在使用了一段时间之后，你很快就发现 Path 类在被创建之后，我们马上就需要规范化这个类。为了简化 Path 的使用，决定在版本 2 中自动规范化这个类：

```
class Path {     //版本 2，自动规范化这个类。
public:
    Path(const char* p) {   /*...*/
            canonicalize();
    }
private:
    void canonicalize();
    //...
};
```

你还可以把 canonicalize 函数声明为 private 接口函数，因为用户根本不需要调用这个函数。

a. 理论上，这个改变是源代码兼容的吗？实际上，这个改变是源代码兼容的吗？如果上面两个都是，那么这个改变是链接兼容的吗？

b. 假设在版本 2 中你把 canonicalize 函数声明为 public 类型，而且对一个已经规范化的 Path 类，调用 canonicalize 函数是无大碍的；那么这个改变理论上是源代码兼容的吗？实际上呢？如果都是，那么是链接兼容的吗？

c. 如果你喜欢将来的版本把 canonicalize 函数声明为 private 类型，那么在版本 2 应该做些什么呢？

7.2　对于程序库的下面每个改变，给出一个用户代码例子；在这个例子中，这个改变是

源代码不兼容的。另外，对于这些改变，在实际中哪些是源代码兼容的？

a. 重新排列类的成员变量。

b. 在类的成员变量中，改变数组元素的数量（即数组的大小）。

c. 在类的声明中，改变这个类的基类的出现顺序（即改变基类的前后位置）。

7.3 下面对类的每个假定的改变，请说明这些改变都是源代码不兼容的：

a. 改变成员函数的返回类型。

b. 添加一个转型函数。

c. 把从基类的公共派生，改变为私有派生。

7.4 对一个程序库而言，下面的哪些改变是源代码兼容的？哪些又是链接兼容的？

a. 在不改变类大小的前提下，改变类的一个私有成员变量的类型。

b. 改变函数参数的默认值。

c. 把一个非虚派生改变成虚派生。

d. 把一个虚派生改变成非虚派生。

e. 添加一个友元声明。

f. 在类的 private 声明部分，添加一个嵌套类。

g. 添加一个纯虚函数的定义。

h. 在不改变添加、删除或者更改任何成员函数实现的前提下，把一个抽象类改变成为非抽象类。假设类中所有的纯虚函数都已经定义为非内联函数，对类定义的改变就是，对于每个之前定义的纯虚函数，删除纯虚说明符（=0）。

i. 在不添加、删除或者更改任何成员函数的前提下，把一个非抽象类改变成抽象类。就是，对于类中我们之前定义的每个函数，都增加一个纯虚标志符（=0）。

7.5 在 7.5 节中，我们说过：对于任何没有重载或者隐藏已存在函数的新函数，增加这个新函数不会破坏链接兼容性。那么，为了不破坏链接兼容性，这个新函数为什么必须不能隐藏已存在的函数呢？这个新的函数又为什么不能重载已存在的函数呢？

7.6 对于类的某些函数，如果它们没有被显式定义，那么编译器将会自动生成这些函数。

a. 对于下面显式定义的函数，哪些函数可以添加到类中，而不会破坏源代码兼容性呢？

- 默认构造函数；
- 析构函数；
- 拷贝构造函数；
- 赋值运算符；
- new 操作符（不含任何额外参数）；
- delete 操作符。

b. 在程序库保持源代码兼容性的前提下，a 部分列出的哪些函数可以从程序库的类中删除呢？

c. 程序库是否可以添加这些函数的显式定义，并且保持链接兼容性？如果当用户编译程序库以前版本的时候，用户的编译系统已经内联了某个函数的默认版本，那么又会出现什么情况呢？

d. 程序库是否可以删除其中的某个函数，而继续保持链接兼容性？

7.7 (*)假设 C++程序库包含了一个函数 might_throw，它用于抛出或者传播(propagate)异常。在程序库的版本 1 中，might_throw 的声明如下：

```
void might_throw() throw(User_blunder, System_oops);
```

另外，你还需要考虑下面几个方面：

1. 某些用户不捕获异常。

2. 某些用户希望处理所有可能被抛出的异常，并且有如下处理程序：

```
try {
    //...
}
catch(User_blunder) {
    //...
}
catch(System_oops) {
    //...
}
```

3. 某些用户希望用这个处理程序来捕获所有的异常：

```
try {
    //...
}
catch(...) {
    //...
}
```

a. 假设在程序库的版本 1 中，might_throw 函数只抛出两个异常 User_blunder 和 System_oops。假设在程序库的版本 2 中，might_throw 函数还抛出一个 Library_glitch 异常，而 might_throw 的异常处理规范（指 try...catch 那一块代码）并没有改变（即没有增加捕获 Library_glitch 异常），那么将会对用户产生什么样的影响呢？

b. 仍然假设在程序库的版本 1 中，might_throw 函数只抛出两个异常 User_blunder 和 System_oops。而在程序库的版本 2 中，might_throw 函数还抛出一个 Library_glitch 异常，且这次 might_throw 的异常处理部分增加了处理 Library_glitch 的规范，那么这样还会对用户产生影响吗？

c. 假设在程序库的版本 1 中，Library_glitch 异常有时会被传播到 might_throw 函数中；从而，如果在程序库的版本 2 中，might_throw 抛出一个 Library_glitch 异常，并且增加了处理 Library_glitch 异常的规范，那么会对用户产生影响吗？

d. 假设在程序库的版本 2 中，might_throw 的异常规范删除了 System_oops；那么如果 System_oops 异常传播到 might_throw 函数的话，会对用户产生什么样的影响呢？

e. 再次假设在程序库的版本2中，might_throw 的异常规范删除了 System_oops，那么要是函数 might_throw 本身不会抛出 System_oops 异常，这个异常也不会从别处传播到 might_throw 中，这样会对用户产生影响吗？

f. 从 a 部分到 e 部分所讨论的情况中，你认为哪些改变是源代码兼容的呢？

7.8 （*）假设你计划提供一个容器类的集合：Set、List、Ordered_list 和 Queue 等等。每个容器类都有：add、remove 和 next 函数。你相信用户完全可以忍受虚函数调用的运行时间开销，因此把这些公共函数都移到一个公共容器基类里面：

```
template<class T>
class Container_int {
public:
    virtual void add(const T&) = 0;
    virtual void remove(const T&) = 0;
    virtual void next(const T&) = 0;
};
```

所有的容器类都继承自 Container_int：

```
template<class T>
class Set : public Container_int<T> {
    //...
};
//其他相似的容器类。
```

你发布了程序库的第1个版本，并开始接收从你基类派生自己容器类的用户的反馈信息。有时，用户会认为：他们设计的某些类根本就不需要 remove 函数，他们抱怨总是被迫编写如下代码：

```
template<class T>
class User_class_with_no_remove : public Container_int<T> {
public:
    void remove(const T&) { /* 输出一个错误消息 */ }
    //...
};
```

毫无疑问，用户宁愿在编译时得到一个诊断消息，也不愿意在运行时得到这个诊断消息。

令人苦恼的是，你发现用户的想法是正确的——有许多容器类不需要提供一个移除（remove）操作。于是，现在你开始用更加精细的体系（见3.4.7节）来设计容器类，如下所示：

```
template<class T>
class Container_int {
public:
    virtual void add(const T&) = 0;
    virtual void next(const T&) = 0;
};
template<class T>
class Container_with_remove_int : public Container_int<T> {
```

```
public:
    virtual void remove(const T&) = 0;
};
template<class T>
class Set : public Container_with_remove_int<T> {
    //...
};
//其他相似的容器类。
```

a. 如果程序库的类体系改变成上面这样，用户将会遇到哪些兼容性问题呢？程序库的哪些用户才会被这个改变所影响呢？

b. 你也可以把设计改变成这样：

```
template<class T>
class Container_with_no_remove_int {
public:
    virtual void add(const T&) = 0;
    virtual void next(const T&) = 0;
};
template<class T>
class Container_int:
    public Container_with_no_remove_int<T> {
public:
    virtual void remove(const T&) = 0;
};
template<class T>
class Set : public Container_int<T> {
    //...
};
//其他相似的容器类。
```

如果对类体系作了上面的改变，那么程序库的哪些用户会遇到兼容性问题呢？

c. 如果从满足那些抱怨的用户的角度考虑，a 部分和 b 部分讨论的改变哪个更好呢？为什么？

d. 暂且不考虑兼容性，为什么这个设计不好呢？

7.9 （*）[ES90]中提出了一种有利于链接兼容性的方法：在类的声明中，先声明类的公共（public）数据成员，然后是保护（protected）数据成员，最后才是私有（private）数据成员。目的在于对那些只使用公共数据成员的代码，增加或者更改私有数据成员或保护数据成员就不需要重新编译这部分代码；而且对那些只使用保护数据成员和公共数据成员的代码，增加或者更改私有数据成员就不需要重新编译这部分代码。

a. 即使数据成员的声明次序如上，但内联成员函数仍然会妨碍所提供的链接兼容性，请解释原因。这个问题是否只局限于公共或者保护类型的内联成员函数呢？

b. 改变类的数据成员经常会改变这个类对象的大小；那么，在目标代码中，哪一部分和对象的大小密切相关呢？

7.12 参考文献和相关资料

Dorward、Sethi 与 Shopiro 的[DSS90]和 Coplien 的[Cop92]中的 9.4 节都提供了有关的、当程序正在运行的时候，如何升级这个程序的更多信息。

第8章

继承体系

如果你真心希望拥有别人的优点，那么就继承并且喜爱这些优点吧。
——Goethe

在这一章里，我们将定义继承体系的根数目、深度和扇出数（fanout）。和某些作者的建议不同的是，我认为：一个继承体系的根数目、深度和扇出数（fanout）是否合适，要取决于这个体系要模拟的领域（domain）和这个体系所需要的特性（property）。接下来，我们讨论几种不同类型的继承体系。最简单的继承体系往往就是效率最高的继承体系，通常也是最好的选择。为了提高扩展性和链接兼容性，稍微复杂的继承体系可以使用接口类（interface class）、对象工厂（object factory）和句柄类（handle class）。最后，我们讨论基于模板的设计和基于继承的设计之间的相互比较。虽然基于继承的设计有它自身的用途，但现实中还是使用得太多了，以致于在那些用基于模板设计可以得到更好效果的地方，还是大量地使用了基于继承的设计。

8.1 根数目、深度和扇出数

考虑一个有向图，它的节点就是程序库为用户提供的类，它的边表示派生关系（就是说，图中有一条从节点 x 到节点 y 的边，代表 x 描述的类 X 继承自 y 节点描述的类 Y，即 Y 是 X 的基类）。在这里，我们把这个图描述的体系称为程序库的继承体系。

某个类如果没有基类，即不是任何类的子类，那么在图中这个类就是根节点。而某个类的深度是指：从根节点到该类最长的可能派生路径中，所经过的类的数目。类的扇出数是指：在给定的继承体系中，从这个类直接派生的类的数目，即该类的直接子类的数目。继承体系的根数目等于根节点的数目；继承体系的深度等于体系中类的最大深度；继承体系的扇出数

等于体系中类的最大扇出数。

某些善意的作者给出了如表 8.1 所示的继承体系设计规则。很明显，表中前两个设计规则是互相矛盾的；也并不是所有的规则都是有根有据的。实际上，在这些规则中，还没有一条规则是放之四海而皆准的。相反的是，给定程序库继承体系的根数目、深度和扇出数的最佳值，要取决于程序库希望模拟的领域（domain）和程序库希望拥有的特性（如扩展性、效率等）。对于某些程序库而言，单根节点体系是最好的；而对于其他程序库而言，多根节点体系就可能是最好的；其次，对于某些程序库而言，深度大或者扇出数大也可能是最好的设计。

表 8.1　　缺乏根据的继承体系设计规则

1. 单根节点体系是最好的。
2. 多根节点体系是最好的。
3. 体系的深度应该不大于 7+2。
4. 体系的扇出数不应该大于 7±2。

例如，假设我们在设计一个字体类型的程序库（16-point Time Italic 就是字体的一个实例）。或许最好的设计就是提供一个公共基类 Font，让所有的字体类都直接或者间接地派生自这个基类；于是，Font 的扇出数就至少等于字体家族的数量（Time 就是一个字体家族实例，字体家族是相似字体组成的集合体）。例如，为了使我们的程序库可以符合任何对扇出数有限制的要求，我们可能就会限制（可以提供的）字体家族的数目，而这有时又是毫无意义的。

让我们来考虑一个更加复杂的例子，假设我们在设计一个程序库，它的类用于模拟各种 C++语言的结构——这包括 if 语句、for 循环、函数定义、constructor initializer（构造函数初始化器，在下面的类定义里，i(0)就是一个构造函数初始化器）等等：

```
class X {
public:
    X() : i(0) { }
private:
    int i;
};
```

图 8.1 给出了这个程序库合理体系的一部分。其中，Construct 描述任何 C++语言的程序设计。C++语言结构的其中一种就是语句(Statement)；语句分为声明语句（Decl）和表达式语句(Expr)。表达式又分成 3 种：if 语句（If）、for 循环语句（For）和构造函数初始化器（Ctor_init）。声明要么是类型声明（Type_decl）——typedef(Typedef)或者类（Class），要么就是非类型的声明（Nontype_decl）。Nontype_del 可以是定义（Defn），也可以是函数声明（Function_defn）。而定义（Defn）可以定义数据（Data_defn），也可以定义函数(Function_defn)。并且函数定义（Function_defn）同时也是函数声明（Function_defn）。函数定义又分为两种：构造函数定义（Ctor_defn）和析构函数定义（Dtor_defn）。这个简单的单根体系的深度虽然只是 7，但这个程序库真实完整的体系肯定会是更深的。对于这个领域而言，单根并且一般深度的体系就是

一个很好的设计。

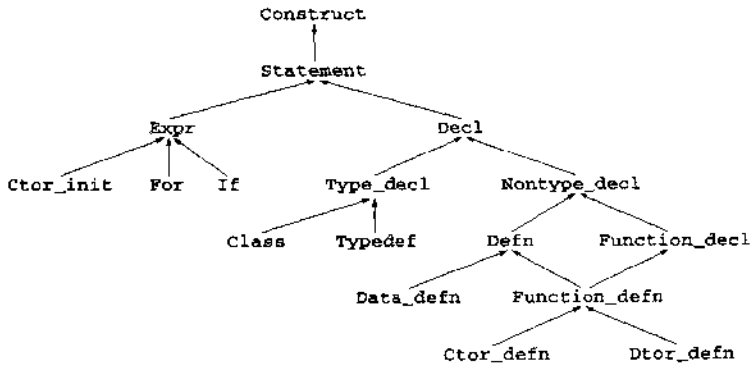


图 8.1　一个 C++语言结构体系

如果被体系深度限制规则所误导，我们可能会通过消除某些派生，来压缩我们的整个体系。而且，两个类（Ctor_defn 和 Dtor_defn）看起来可以是删除的对象。假设我们删除了这两个类，而只用类 Function_defn 来模拟所有的函数定义。进一步假设我们最初的深度设计提供了下面的函数，它返回构造函数初始化器（constructor initializer）的集合（其中使用了一个描述集合的类 List）：

```
class Ctor_defn {
public:
    List<Ctor_init*> initializers() const;
    //...
};
```

当我们删除了 Ctor_defn 类之后，就必须把 initializers 函数移进 Function_defn 类里面：

```
class Function_defn {
public:
    List<Ctor_init*> initializers() const;  //把上面的函数移到这里.
    //...
};
```

现在就必须决定：由于 initializers 并不是一个构造函数（constructor）定义，那么对 Function_defn 对象调用 Function_defn::initializers 应该做些什么呢？我们可以让这个函数返回一个空的 List，或者还可以给 initializers 函数一个前提条件：它只能被除了构造函数（即 Ctor_defn）定义以外的任何定义调用。究竟选择哪种设计，要取决于当用户使用一个非构造函数（constructor）定义的函数定义调用这个函数的时候，哪种选择是合法的。和我们最初的、更深的体系相比，这种设计还会产生编译时错误。因此，这种希望减少体系深度的设计，将会使我们陷入一种类型不安全的困境：也就是说，最初的、更深的设计更好。

现在假设我们还希望在程序库中提供一个类 Parser，它描述一个 C++的解析器：

```
class Parser {
public:
```

```
    Constructor* parse(const char* code);
    //...
};
```

函数 parse 创建了一棵描述 C++代码的树：这里的代码是指以 null 结尾的字符串 code，树中的每个节点是 Construct 体系（见图 8.1）中类的实例。函数 parse 将返回一个指向新建树根节点的指针。

于是，我们的程序库现在就具有两个根——Construct 和 Parser。假设我们为了遵从表 8.1 的第 1 条规则，希望通过改变类的体系来使程序库变成单根体系。为了达到这个目的，我们可以让 Construct 和 Parser 共同派生自一个公共基类：让我们暂且把这个公共基类称为 Construct_or_Parser。因为 C++语言结构和 C++解析器是两种完全不同的事物，所以 Construct_or_Parser 将几乎不会有什么数据成员，甚至还可能没有任何非 nice(nice，见 2.3 节)成员函数。因为程序库的用户应该不会编写一个可以多态地引用一个 Construct 或者 Parser 为参数的函数，所以 Construct_or_Parser 实际上是没有用途的——只是增加了程序库的复杂度。因此，对于我们程序库的这个版本，多根体系要比单根体系优越。

8.2 体系类型

可重用程序库的设计可以基于几种继承体系类型其中的一种，也可以是这些类型的组合。其中，直接类型是最高效而且最简单的体系类型；而与直接类型相比，接口体系（Interfaced hierarchie）、结合对象工厂（object factory）的接口体系和句柄体系都有很多优点——很明显，它们都提高了扩展性和链接兼容性——但它们也有某些缺点。

8.2.1 直接体系

假设我们在设计一个消费电器程序库，它主要用于提供描述下面电子产品的类：电视（TV）、压缩唱片播放器（CD）、个人电脑（PC）等等。假设我们程序库的所有电子产品都有一些相同的函数——它们都有一个序列号并且都能被打开。于是，我们程序库最简单、最直接的设计就是：让这些电器设备类都继承自一个公共基类 CED（consumer electronic device 的缩写）：

```
class CED {
public:
    string serial_number() const { return _serial_number; }
    virtual void turnon() = 0;
    //...
private:
    string _serial_number;
    //...
};
```

对于所有的电器设备而言，serial_number 函数的实现都是一样的。

下面是 TV 的实现：

```
class TV : public CED {
public:
    void turnon();
    void set_channel(int channel);
    int current_channel() const;
    //...
};
```

类 TV 实现了 CED 中声明为纯虚函数的 turnon 函数，并且提供了一些特用于电视的函数（set_channel、current_channel 等）。类 CD 和 PC 的实现和这个实现相似。

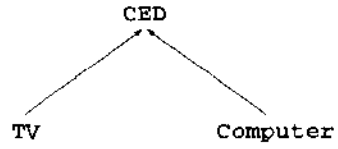


图 8.2　消费电器程序库的继承体系

图 8.2 给出了消费电器设备继承体系的部分内容，这种设计方法既简单而且容易理解，又要比我们接下来要讨论的几种设计类型具有更高的效率。然而，它有两个缺点。首先，它不能像 3.2.1 小节讨论的接口类那样，具有很好的扩展性；其次，我们将在 7.5 节看到，对类的实现的改变往往是链接不兼容的。

在下面 3 节里，我们将给出其他的一些（体系类型）设计，它们提高了扩展性和链接兼容性，但这些都是以增加复杂度和降低效率为代价的。一如既往，对一个特定程序库而言，最好的设计要取决于程序库用户的需要。

8.2.2　接口体系

我们在 3.2.1 小节看到，接口类可以提高程序库的扩展性。图 8.3 也给出了我们程序库体系的接口体系版本：

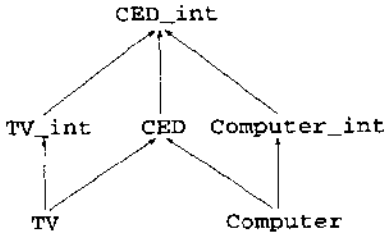


图 8.3　消费电器程序库的接口版本

下面是接口类 CED_int：

```
class CED_int {
public:
    virtual String serial_number() const = 0;
    virtual void turnon() = 0;
    //...
};
```

下面是接口类 TV_int：

```
class TV_int : public virtual CED_int {
public:
    virtual void set_channel(int channel) = 0;
    virtual int current_channel() const = 0;
    //...
};
```

函数 serial_number 的公共实现定义在非接口类 CED 里面：

```
class CED : public virtual CED_int {
public:
    String serial_number() const { return _serial_number; }
    //...
private:
    String _serial_number;
    //...
};
```

于是，用户如果希望创建一个实现完全不同于 TV 的另外一个描述电视的类，那么用户现在就可以直接继承自 TV_int 类，来满足自己的需要。

除了增加扩展性以外，接口类同时也增加了链接兼容性。可以看到，对于那些只使用 TV_int 类、而不直接使用 TV 类的代码，我们可以改变 TV 类的实现，而不会破坏用户代码的链接兼容性。幸运的是，完全使用 TV_int 来编写程序是有可能的。例如，改变频道的函数 surf，就可以只使用 TV_int 类来实现：

```
void surf(TV_int& tv) {
    int i = tv.current_channel();
    while(!showing_program_viewer_likes(tv))
        tv.set_channel(++i);
}
```

只有当程序员需要创建一个新的 TV 对象时，才需要直接使用到 TV 类：

```
TV_int* new_TV() {
    return new TV;  //TV, 而不是 TV_int.
}
```

为了可以尽可能简单地升级到程序库的新版本，程序员应该把使用非接口类的所有代码隔离成尽可能少的几个文件，那么当程序库升级的时候，重新编译的就只是这几个文件而已。

如果体系中的每个类都是接口类，我们就称这个体系是接口体系。如果程序库体系是接口体系，那么用户就可以使用接口类来编写几乎所有的代码；只有当需要创建一个程序库类

的实例时，程序员才会必须使用非接口类。

与非接口体系相比，接口体系也有几个缺点。首先，接口体系要比非接口体系复杂。那些不关心扩展性和链接兼容性的程序员将不愿意增加这些复杂性。其次，接口体系的实现比较困难。再次，接口体系总会带来许多虚派生和虚函数。例如，图 8.3 中所有从类 CED_int 的派生和函数 serial_number 都是虚的。而在 8.2.1 小节的直接体系中，这些派生都不是虚派生，函数 serial_number 也不是虚函数。就效率而言，虚派生和虚函数分别比不上非虚派生和非虚函数。

8.2.3　对象工厂（Object Factory）

从上一小节我们看到：在继承体系中，只有对那些不需要创建类实例的代码，接口体系才能提供链接兼容性。为了使程序库用户可以编写这样的程序：当需要升级到程序库的新版本时，根本就不需要重新编译任何代码；我们可以给我们的接口体系提供一个 object factory（也称作 kit）。下面就是用于消费电器程序库的 object factory：

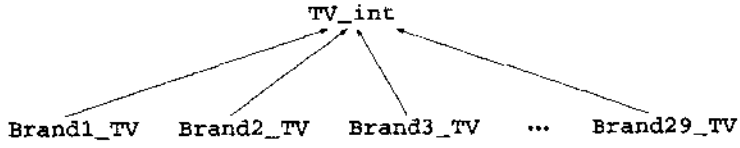
```
class CED_factory {
public:
    static TV_int* new_TV();
    static PC_int* new_PC();
    //...
};
```

每个 factory 函数的参数列表通常和相应类构造函数的参数列表是一样的。而且，factory 函数的定义必须是外联的：

```
TV_int* CED_factory::new_TV() {
    return TV;
}
PC_int* CED_factory::new_PC() {
    return new PC;
}
```

如果只针对这些只使用程序库的 object factory 和接口类的程序，那么任何对程序库类实现的改变都是链接兼容的。

除了增加链接兼容性以外，对象工厂还有另外两个优点。首先，它简化了程序库展现给用户的继承体系。假设我们程序库的部分继承体系看起来如下所示：



如果使用 factory 函数来创建所有种类的电视类，那么我们就可以把所有的 BrandX_TV

类都移到我们程序库的实现内部：

```
class CED_factory {
public:
    enum TV_brand { Brand1,Brand2, /*...*/ ,Brand29 };
    static TV* new_TV(TV_brand brand);
};
```

许多用户都认为，与具有许多类集合的程序库接口相比，具有单一枚举类型的接口显得比较简单。

对象工厂还可以用来隐藏某些实现决策。假设我们希望提供两种电视：高精度电视和低精度电视。于是，我们就有两种设计方案。第一个设计中，只用一个类的实例就可以描述两种精度的电视：

```
class TV : public virtual TV_int {
public:
    TV(bool high_density = false);
    //...
};
```

在第二种设计中，我们用不同的类来描述高精度的电视和低精度的电视：

```
class HDTV : public virtual TV_int {
public:
    //...
};
class LDTV : public virtual TV_int {
public:
    //...
};
```

无论我们选择哪一种设计，通过提供一个 factory 并把所有非接口类的实现都移到我们程序库的实现中去，都可以让我们的决策不被程序库用户所知：

```
class CED_factory {
public:
    static TV_int* new_TV(bool high_density = false);
    //...
};
```

只看到接口类和 factory 的用户将不能判断高精度电视和低精度电视究竟是由一个类来实现的呢，还是由两个类来实现的（即看不到接口类的实现细节）。因此，如果有必要的话（即知道实现细节的必要），在程序库将来的版本里，应该可以改变高精度电视和低精度电视的实现细节，至少应该给自己提供改变的灵活性。

然而，只给程序库用户提供接口类和 factory（即并没有提供非接口类）也有几个缺点。因为用户现在如果需要创建程序库对象，就只能是通过 factory 来创建；而这些通过 factory 创建的对象只能是存储在空闲存储区域。与存储在静态存储区域或者堆栈相比，把对象存储在空闲存储区域的效率显得比较低，并且使用空闲存储区的对象往往是容易产生错误的（8.3.1 小节详细论述了把对象移到空闲存储区的缺点）。只提供接口类和 factory 的另一个缺点是，

对于那些以前使用程序库某些非接口类来实现他们自己特殊类的程序员，现在由于看不到非接口类的细节，现在也就不可完成以前的实现了。实际上，程序库应该提供某些非接口类，否则（由于省略了这些非接口类）将会使用户代码效率不佳、容易产生错误或者难以实现。

8.2.4　句柄体系

我们已经在 8.2.2 小节和 8.2.3 小节看到，（和其他某些体系一样）接口类和对象工厂都可以用来提供链接兼容性。在这一小节我们将看到，提供链接兼容性的另一种方法是使用句柄类。如果一个类的实例就只包含一个数据成员，并且这个数据成员是指向其他对象的指针，那么我们就称这个类为句柄类。例如，假设我们使用一个句柄来实现 TV 类，而不让它像以往那样从接口类派生：

```
class TV_rep {
    //...
};
class TV {
private:
    TV_rep* rep;     //唯一的数据成员。
    //...
};
```

于是，对类 TV 的用户而言，任何对类 TV_rep 实现的改变都是链接兼容的。而且，程序库文档应该清楚地声明哪个类是句柄类；还应该提醒文档读者，只有对那些仅仅使用句柄或者接口类而没使用任何其他程序库类的代码，进行的改变才能保证是链接兼容的。

一般情况下，句柄类的每个成员函数只是调用所描述类的相应成员函数：

```
class TV_rep {
public:
    int current_channel() const;
    //...
};
class TV {
private:
    TV_rep* rep;
    //...
public:
    int current_channel() const {
        return rep->current_channel();
    }
    //...
};
```

注意，TV_rep::current_channel 和 TV::current_channel 中只有其中一个（而不是两个）能成为内联函数。把这两个函数都声明为内联函数会与我们的目的相悖，因为函数实现的编译代码将会出现在用户的目标文件里面，并且对 TV_rep 实现的改变也会要求用户进行重新编译。

即使把句柄类的所有成员函数都声明为内联函数，句柄类的效率还是比不上非句柄类。从上面代码可以看出，每个对 TV 成员函数的调用实际上是对 TV_rep 相应成员函数的间接调用。

为了满足那些既要求最大效率又要求链接兼容性的用户，我们可以提供两个类：句柄类和描述（representation）类（如 TV_rep）。然而提供两个类将会使程序库的接口更加复杂，并且使程序库本身的维护更加困难。因此，在程序库的实现中，程序库设计者往往会隐藏描述类。

当我们试图把体系中所有的类都实现为句柄类的时候，句柄类就显得非常有意思了。假设我们提供了图 8.4 所示的体系，并且我们希望该体系中所有的类都成为句柄类，那么第一件我们需要做的事就是，创建一个如图 8.5 所示的描述（representation）体系：

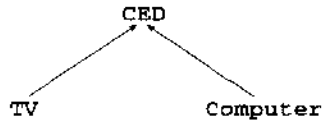


图 8.4　句柄类的体系

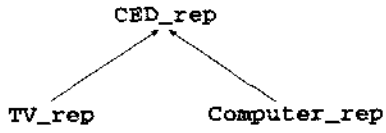


图 8.5　描述类的体系

下面是 CED_rep 类：

```
class CED_rep {
public:
    string serial_number() const {
        return _serial_number;
    }
    virtual void turnon() = 0;
    //...
private:
    String _serial_number;
};
```

下面是 TV_rep 类：

```
class TV_rep : public CED_rep {
public:
    void turnon();
    void set_channel() const;
    int current_channel() const;
    //...
};
```

现在就可以实现我们的句柄体系了，类 CED 将转到类 CED_rep：

```
class CED {
public:
    String serial_number() const {
        return rep->serial_number();
    }
    void turnon() {
        rep->turnon();
    }
    //...
protected:
    CED(CED_rep* r) : rep(r) {}
    CED_rep* rep;
};
```

(请牢记：CED::serial_number 或者 CED_rep::serial_number 中，至少应该有一个必须是外联函数。)虽然 turnon 函数在描述体系中是虚函数，但在句柄体系中，它并不需要实现为虚函数；因为对于任何句柄类而言，上面给出的 CED::turnon 函数的实现都是正确的。另外，由于 CED_rep 是一个抽象类，所以 CED 并没有提供 public 构造函数；所提供的 protected 构造函数主要是用来帮助构造派生类对象。

下面是 TV 类的实现：

```
class TV : public CED {
public:
    TV() : CED(new TV_rep) {}
    void set_channel(int channel) {
        ((TV_rep*)rep) >set_channel(channel);
    }
    //...
};
```

我们在转调 set_channel 函数之前，必须向下转型 rep。因为 CED_rep 是 TV_rep 的非虚基类，所以这种转型是合法的；并且这种转型也是安全的，因为 TV 的构造函数初始化器已经把 rep 初始化为一个 TV_rep 对象了。(任何 TV 的派生类都应该把 rep 转型为 TV_rep 类型或者 TV_rep 的派生类类型。)如果 CED_rep 是 TV_rep 的虚基类，那么我们就必须使用练习 8.9 所讨论的向下转型机制了。

就扩展性而言，句柄体系比不上直接体系或者接口体系（见练习 8.10）。为了在句柄体系里提供更好的扩展性，我们可以增加一个接口类，如图 8.6 所示。这个体系和图 8.3 所示的体系看起来是相同的，惟一的重大区别在于：在图 8.3 中，非接口类都不是句柄；而在图 8.6 中，非接口类都是句柄。为了给图 8.3 体系的用户提供完整的链接兼容性，我们还需要提供一个对象工厂，从而对那些注重链接兼容性的用户，就可以通过 factory 只在空闲存储区创建类的实例。另一方面，图 8.6 的体系提供了一个不具备对象工厂的链接兼容性；于是，用户就可以在任何存储空间（静态存储区、堆栈存储区或者空闲存储区）创建他们的类实例（虽然底层的表述对象是位于空闲存储区，但这些都是实现细节，对用户是不可见的）。因此，对那些注重链

接兼容性的用户而言，接口化的句柄体系将要优于具有 factory 的接口（非句柄）体系。

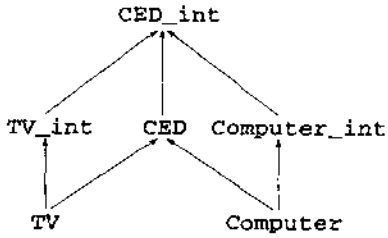


图 8.6　图 8.4 的句柄体系的接口化版本

然而，与具有 factory 的接口（非句柄）体系相比，接口化的句柄体系具有几个缺点：首先，接口化句柄体系的实现更加困难。对一个现实的程序库而言，设计和实现句柄和 representation（描述）两个体系往往是乏味而且浪费时间的。句柄体系设计的维护也更加困难，因为必须将所做的改变应用到两个不同的体系中。

其次，接口化的句柄体系虽然比非接口的句柄体系具有更好的扩展性；但与具有 factory 的接口（非句柄）体系相比，它的扩展性将处于下风。例如，假设我们的用户希望模拟一台新类型的 PC，它具有数学协处理器，称为 Math_PC，并且 Math_PC 可以重用 PC 实现的大多数代码。于是，在具有 factory 的接口（非句柄）体系中，我们只需要提供一个可继承的 PC 类，并让 Math_PC 直接从 PC 继承就可以了；而在接口化的句柄体系中，我们必须设计 PC 和 PC_rep 这两个可继承的类（PC 只是一个简单的句柄类，或许也并不需要费多大劲设计成可继承的类），而且，我们还必须给用户提供句柄和 representation（描述）两个体系，然后用户必须让 Math_PC 派生自 PC，Math_PC_rep 派生自 PC_rep，并且确定所有的转调函数都可以正确地实现。

8.3　模板还是继承

模板和继承是 C++的两个强大特性。遗憾的是，对于类体系的设计究竟应该基于模板还是基于继承这个问题，C++程序员却常常使感到疑惑，甚至混淆不清。在可以使用模板来获得更佳设计的时候，程序库的设计者有时却会不明智地使用继承。

考虑容器类的设计。在这整本书里，我们在设计容器类的时候，都使用模板参数来指定容器所包含值的类型。如 2.6 节和 5.1 节我们设计的 Stack 模板：

```
template<class T>
class Stack {
public:
    void push(const T& t);
    T pop();
    //...
};
```

另一方面，很多人使用的设计容器类的方法（特别是在模板成为 C++语言的组成部分之前）是提供一个抽象基类，然后让元素类型从这个基类派生出来。例如，下面是一个堆栈类，它要求它的元素类型派生自抽象基类 Stackable：

```
class IBStack : public Stackable {
public:
    void push(Stackable* s);
    Stackable* pop();
    //...
};
```

在这个讨论中，为了避免和基于模板的类 Stack 发生混淆，我们把我们的类命名为 IBStack(inheritance-based stack 的缩写)。函数 push 在 IBStack 的顶部插入值 s；函数 pop 将会移除栈顶元素的值，并返回这个值。之所以让 IBStack 派生自 Stackable，是因为用户可能希望创建元素类型为 IBStack（实际上应该是指向 IBStack 的指针）的 IBStack[1]（基于继承的堆栈）。于是，如果用户希望创建某个类型 Widget（实际上是指向 Widget 的指针）的 IBStack（基于继承的堆栈），那么就应该让 Widget 从 Stackable 公共派生出来：

```
class Widget : public Stackable {
    //...
};
```

然而，这些基于继承的堆栈有几个严重的缺陷，而基于模板的堆栈并没有这些缺陷。在下面的章节里，我们将讨论 IBStack 的这些缺点，而把后一个问题（即基于模板的堆栈并没有这些缺点的原因）留给读者来分析（见练习 8.11）。

8.3.1　指针操纵

基于继承的容器的第一个缺点就是需要用户自己操纵指针（IBStack 的 push 和 pop 操作分别接收和返回指针）。遗憾的是，指针操纵是很容易引起错误的；让我们考虑下面的代码：

```
IBStack create_stack() {
    IBStack s;
    Widget w;
    s.push(&w);
    return s;        //oops
}
```

create_stack 函数返回的 IBStack 对象 s 将会包含一个曾指向 w 对象的无效指针。与所包含的指针元素所指向的对象的生存期相比，IBStack 的存活期应该不长于这些对象的最短生存期，所以上面的 IBStack 对象 s 同样也就是无效的。于是，对于上面这样的程序库，要让用户不犯生存期错误是很困难的。

为了减少创建无效 IBStack 对象的风险，许多用户可能会只插入在空闲存储区分配的对

[1] 译注：这个 IBStack 和前面的 IBStack 是不同的。这个 IBStack 是堆栈，可视为容器类；前面的 IBStack 是堆栈的元素，可视为元素类。

象。然而，这种做法却会强迫用户把所有本来应该存储在静态存储区或者堆栈存储区的对象，都移入空闲存储区。例如，这些用户可能会这样改写 create_stack 函数：

```
IBStack create_stack() {
    IBStack s;
    Widget* w = new Widget;
    s.push(w);
    return s;    //ok
}
```

而且，把对象都移入空闲存储区将会增加内存泄漏的风险。例如，这个 create_stack 版本的所有用户都必须记住：（最终）要删除返回堆栈指向的所有对象（如 w 所指向的对象）。因此，对于那些要操纵许多基于继承的容器的程序库实现者而言，在防止内存泄漏方面将会遇到一些麻烦。

8.3.2 派生要求

假设一个用户希望创建一个整数 IBStack，并且需要执行下面的操作：

```
int* i;
IBStack int_stack;
int_stack.push(i);        //错误
```

遗憾的是，上面的代码是非法的，因为 int 并不是从 Stackable 派生的。为了操纵一个整数堆栈，用户必须编写一个新类：

```
class Int : public Stackable {
    int val;
    //...
};
```

现在用户就可以创建一个 Int 的 IBStack 了：

```
Int* i = new Int;
IBStack int_stack;
int_stack.push(i);
```

然而，这个解决方案说起来容易，做起来很难。编写一个模拟内建 int 类型的 Int 类是相当困难的（见练习 2.8）。不仅如此，编写一个 Int 可能仅仅只是一个开始，用户可能还需要自己编写 Bool、Char、Short、Long、Float 和 Double。

上面必须派生自 Stackable 的要求，不仅会给内建类型带来问题，也会给用户定义类型带来许多问题。显然，并不是每个类都可以派生自 Stackable 的。许多类的设计者可能从没听过 Stackable，许多知道 Stackable 的设计者也可能不会让他们的类从 Stackable 继承。例如，使用其他 C++程序库的设计者可能不希望用我们的程序库来编译他们的程序库，因此也就不会让他们的类派生自 Stackable。

考虑一个类 Fidget，它不是从 Stackable 派生的。于是，IBStack 的用户有两种方法可以实现一个 Fidget 堆栈（即元素为 Fridget 对象的堆栈）。第一种方法就是改变 Fidget 的声明，如下所示：

```
class Fidget : public Stackable {        //添加一个派生声明。
    //...
};
```

当然，在不用其他方式改变派生类（如 Fidget）的接口和实现的前提下，要为给定的类（指 Fidget）增加一个新的基类（指 Stackable）几乎是不可能的（因为接口肯定会因为添加基类而增多）。

即使程序员知道如何改变 Fidget，但如果程序员没有存取 Fidget 源代码的权限，那么增加一个新的基类也是不可能实现的。于是，这类程序员只能实现第二种方法——派生出 Fidget 的 Stackable 版本：

```
class Stackable_fidget : public Fidget, public Stackable {
    //...
};
```

虽然从理论上讲，这个做法是可行的，但是考虑到一个程序员如果使用了 5 个程序库，每个程序库都包含了和 Stackable 类似的容器基类，而且每个程序库都至少有 10 个类需要作为容器的元素，可以插入到其他程序库的容器中；那么，用户面前要实现的任务是令人畏缩的，而且在大多数系统上，多重继承的使用也会导致产生许多低效的代码。

8.3.3　实现不需要的函数

考虑一个提供了几个基于继承的容器类的程序库，这些容器类包括 IBStack、IBSet 和 IBSortd_list。为了使用户需要派生的类的数量是可以控制的（见练习 8.12），程序库设计者可能会为所有的容器类实现一个共同的抽象基类，暂且把这个基类命名为 Containable：

```
class IBStack : public Containable {
public:
    void push(Containable*);
    //...
};
class IBSet : public Containable {
public:
    void insert(Containable*);
    //...
};
```

假设 IBSet 要求被插入对象具有一个 hash（哈希）函数，IBSorted_list 要求被插入对象具有一个 compare（比较）函数，其余的容器类类似。那么，这些函数就必须在 Containable 内部声明：

```
class Containable {
public:
    virtual long hash() const = 0;
    virtual int compare(const Containable*) const = 0;
    //...
};
```

但现在，如果用户需要在 IBStack 中插入 Widget 对象，那么这个用户就必须定义

Widget::hash 和 Widget::compare 函数，甚至其他的一些函数——即使 IBStack 并不需要这些函数，这个过程也不能少。对于大多数用户而言，当他们被迫实现他们程序中并不需要的函数时，可能会采取下面比较简单的方法：

```
class Widget : public Containable {
public:
    long hash() const {
        assert(0);
        return 0;
    }
    int compare(const Containable*) const {
        assert(0);
        return 0;
    }
    //...
};
```

遗憾的是，如果偶然地调用了这些函数，将会出现一个运行时错误，而不是编译时错误。因此，程序库不应该要求用户实现任何不需要的函数。

8.4 总结

在一个继承体系中，根数目、深度和扇出数是否适当，要取决于这个体系要模拟的领域和希望具有的特性。

虽然直接继承体系既是最容易实现的类型，又是最高效的类型，但是接口体系、对象工厂和句柄体系都有利于程序库版本之间的链接兼容性。而且，接口体系还有助于增强程序库的扩展性。表 8.2 总结了 8.2 节所讨论的几种体系类型之间最重要的区别。一如既往，并没有哪一种体系类型对所有程序库都是最好的；程序库设计者必须针对自己的程序库和用户类型，来选择最适合的体系类型。

表 8.2　　各种继承体系类型的优缺点

继承体系结构	复杂性	效率	扩展性	链接兼容性
Direct	simple	最大的	一般	minimal
Interfaced	complex	降低	good	部分
Interfaced+Factory	complex	降低	good	total
Handle	simple	降低	poor	total
Interfaced Handle	complex	降低	good	total

当使用模板可以获得更佳设计的时候，程序库设计者应该格外小心，防止不明智地使用继承。

8.5 练习

8.1　假设你在设计一个用于多媒体应用程序的程序库——就是说，这个应用程序主要是用来操纵各种媒体，如文本、音频、视频和各种同步媒体（同步媒体是指由两种或两种以上媒体通过同步组合后的媒体，例如，带有音响的动画片）。假设多媒体程序库已经拥有这几个类：Medium, Text, Audio, Video 和 Synced；请给出这个程序库的一个直接继承体系。

8.2　现在假设你的多媒体程序库（见练习 8.1）将会提供音频、视频和同步媒体的数字（相对于模拟）实现，并且你的程序库将会用具体的类来描述这些所需要的数字实现。请为现在的多媒体程序库给出一个接口体系。

8.3　练习 8.1 的直接继承体系的一个合理的实现是：把 Audio 和 Video 共同的部分都放入 Medium 里面。现在假设增加了一种新的媒体（aromas），并用 Olfactory 来描述这种媒体。

a. 假设 Olfactory 的实现和 Audio 与 Video 的实现相同的地方很少，那么你将如何把 Olfactory 的数字实现添加到在练习 8.1 设计的直接继承体系中呢？

b. 你应该如何把 Olfactory 的数字和模拟实现都添加到在练习 8.2 设计的接口多媒体体系中呢？

8.4　针对练习 8.2 所给出的接口多媒体程序库，编写一个对象工厂（object factory）。

8.5　如前所述，句柄体系可以是接口类型，也可以是非接口类型。

a. 针对练习 8.1 多媒体程序库，给出一个非接口的句柄体系。

b. 假设声音媒体拥有一个虚函数 volume，它返回声音对象当前的音量大小，并把返回值存储在私有成员_volume 中。在你的声音媒体的句柄类和描述（representation）类中，分别给出 volume 函数的代码体。请声明这个函数是否是 const 类型或者 inline 类型的。

c. 与直接体系相比，非接口化的句柄体系有哪些优缺点？

d. 请接口化在 a 部分所设计的继承体系。

e. 与不具有句柄的接口体系相比，接口化的句柄体系有哪些优缺点？

8.6　假设你在给一个旅行社设计一个网上订单系统。你的继承体系将会包含下面的类，它们都是从 Reservation 派生而来：

```
Car_rental_reservation
Lodging_reservation
Train_reservaton
Limo_reservation
Airline_reservation
Curise_ship_reservation
```

你可以通过使用 object factory 来简化程序库的接口，请给出你的实现方法。

8.7　假设在某些程序库中，你希望模拟两种类型的声音媒体——磁带和 CD——并且有两种可能的设计方案。首先，那些描述磁带和 CD 的对象可以用单个类的实例来描述：

```
class Audio_medium : public Medium {
public:
    Audio_medium(bool tape = false);
    //...
};
```

另一种设计方法是使用不同的类来分别描述磁带和 CD：

```
class Tape : public Medium { /*...*/ };
class CD : public Medium { /*...*/ };
```

如果使用 object factory，你可以隐藏设计决策，不让你的用户知道；请给出你的实现方法。

8.8 对于练习 3.8 中讨论的“复杂构造函数”问题，我们应该如何使用 object factory 来解决这个问题，从而不需要派生出任何新的派生类(如 Child_toothpaste)?

8.9 对于 8.2.4 小节的 TV 类，如果 CED_rep 是 TV_rep 的虚基类，那么 TV 类的向下转型将是非法的。在这个练习里，我们将考虑另外一种设计方法，从而使虚继承不会导致上面的问题。

a. 使用一个 dynamic_cast 结构，如何代替上面所述的转型呢？这种解决方案有哪些缺点呢？

b. 假设{X1，X2，...，Xn}是直接或间接从 CED_rep 派生出来的类集合，考虑下面的函数：

```
class CED_rep {
public:
    virtual X1* cast_to_X1();
    virtual X2* cast_to_X2();
    //...
    virtual Xn* cast_to_Xn();
    //...
};
```

函数 cast_to_Xi 如果是被 Xi 类型的对象或它的派生类型的对象调用，那么将会返回一个指向这个对象的指针，否则将返回 0。在不使用 dynamic_cast 操作符的前提下，给出这个函数的实现方法。并且解释当你的函数是在虚继承下被调用的时候，你的实现仍然是正确的原因。

c. b 部分的设计是可扩展的吗？为什么？

d. 如何把 CED 的描述体系压缩到单个类里面，从而我们就不需要 8.2.4 小节的转型？这个设计方法的两个缺点又是什么？

e. 假设我们使用下面的一对数据成员，来代替 8.2.4 小节的 CED::rep 数据成员：

```
class CED {
private:
    void* rep;
    int reptype;
    //...
```

```
};
```

使用上面的数据成员，我们可以重新设计句柄体系和相应的描述体系，并且在这些体系中可以完全不需要向下转型，如何实现？

f. 在什么条件下，e 部分的设计才是低效的而且不可扩展的？

g. 如果你在实现 CED 程序库，并且 TV_rep 要从 CED_rep 虚派生，那么对于 a 部分、b 部分、d 部分和 e 部分的设计方案，你会选择哪一个？

8.10　在 8.2.4 小节中，我们提出就扩展性而言，句柄体系不如接口体系。为什么？

8.11　考虑 2.6 节和 5.1 节的 Stack 模板。

a. Stack 并没有 8.3 节所讨论的 IBStack 模板的任何缺点，请给出分析。

b. Stack 的哪个缺点是 IBStack 不具有的？

c. Stack 的这个唯一的缺点是否比 IBStack 所有的缺点还要严重？为什么？

8.12　对于 8.3.3 小节所讨论的基于继承的容器类，如果对每个这样的容器类都使用不同的抽象基类，那么在用户代码中，将会有许多不可缺少的派生出现。请给出一些这样的例子。

8.6　参考文献和相关资料

Martin[Mar91]、Nackman 与 Barton[NB94]和 Linton 与 Pan[LP94]都讨论了在 C++程序库中使用接口类和效率低下之间的联系。Linton[Lin92]给出了 object factory 的一个实例。

Gorlen、Orlow 与 Plexico[GOP90]文档化了一个 C++程序库的实例，这个程序库提供一个基于继承的容器类。Carroll[Car95]更加详细地讨论了基于继承的容器类的不足之处。

练习 8.9 是在和 Rich Kempinski、Deborah、McGuinness 与 Elia Weixelbaum 的一次讨论后引申出来的。

第9章

移植性

宝石遗传下来的时间越长，它所携带的价值就越加贵重
—— Lord Stanley

对于一段代码，它的可移植性越好，重用性就越好。在这一章里，我们将讨论影响C++代码可移植性的主要因素：对C++语言定义的状态改变、不确定的行为、实现性定义并且未经指定的行为、实现依赖性、模板实例化（也叫具体化）、运行时程序库和其他一些依赖于操作系统的因素。

9.1 有编写可移植代码的必要吗

对于许多编程项目，我们的关注点并不在于可移植性。然而，对于其他一些编程项目，移植性却是我们主要的目标。众所周知，并不是所有的代码都可以在所有的平台运行的。于是，对于某段代码，如果可以很容易地在某个平台运行，我们就称这段代码对这个平台是可移植的。另外，如果改变这段代码，使它可以在给定的平台运行，我们就把这种行为称为把这段代码移植到这个平台。

当我们设计代码的时候，我们总是（或者应该）对这段代码可以运行的平台的范围略有所知。我们可能只希望这段代码可以在我们开发小组程序员所使用的相似平台运行；我们也可能希望这段代码可以在所有使用Windows操作系统的平台上运行；我们甚至还希望这段代码可以在几个不同种类的、使用不同操作系统的硬件（如PDA、台式机等）上运行。

9.1.1 可移植性的优缺点

可移植性通常会和效率与实现的难易程度相互制约。具体地说来，在一个或者多个平台

上，容易实现的可移植代码通常都是低效的。譬如，假设我们希望清除内存中从位置 p 开始的 n 个字节。可以如下编写代码：

```
memset(p,n,0);
```

对于每个提供了 ANSI/ISO C++程序库的平台，这段代码都是可移植的，因为该程序库已经定义了 memset 函数。现在假设我们要求清除内存的这段代码具有很高的实时性；于是我们可以使用 bzero 函数，它清除内存的速度要比 memset 函数快，但只有某些平台提供这个函数：

```
bzero(p,n); //要比 memset(p,n,0)快。
```

然而，bzero 函数可能并不在 ANSI/ISO C++程序库里面；因此调用 bzero 函数可能会破坏我们代码的可移植性。

为了保留原来的移植性，并且在含有 bzero 函数的系统中体现实时的要求，我们可以如下改写我们的代码：

```
#indef HASBZERO
    bzero(p,n);
#else
    memset(p,n,0);
#endif
```

我们假设某些定义有 HASBZERO 的平台都会提供 bzero 函数。另外，如果我们在好几个地方都调用了 bzero 函数，我们也可以自己编写一个 bzero 函数，让那些没有提供 bzero 函数的平台也可以调用这个函数：

bzero.h:

```
#include<stdlib.h>
#include<memory.h>
#ifndef HASBZERO
inline void* bzero(char* p, size_t n) {
    return memset(p, n, 0);
}
#endif
```

如果所有提供 bzero 函数的平台都把这个函数包含在 stdlib.h 头文件中，那么当我们需要清除内存的时候，就可以如下编写代码：

```
#include<bzero.h>   //bzero.h 头文件已经包含了 stdlib.h 头文件。
//...
bzero(p, n);
```

这段代码不但是可移植的，而且是高效的。然而，与前面只是简单地调用 memset 函数相比，这段代码的实现又是困难的。通常地讲，把实际代码移植到很多平台都会使代码变得复杂，甚至很不优雅。

9.1.2 目标代码和创建过程的可移植性

当考虑可移植性的时候，通常都会想到源代码，但可移植性也能应用于目标代码。明确

地说，对于一段目标代码，如果可以拷贝到某个平台，并且不需修改立即就可以使用，那么这段目标代码对这个平台是可移植的。如 memset 的例子所展现的那样，程序员可以设法提高源代码的可移植性，但是却不能提高目标代码的可移植性。目标代码可移植与否取决于相关平台的类型，而不是产生该目标代码的源代码。

实际上，目标代码是很少具有可移植性的。因此，如果希望在一个新的平台 P 创建我们程序库的目标代码，那么我们通常都必须把源代码先移植到平台 P，然后在这个平台创建这段代码。这个过程要求我们程序库的创建过程（built procedure）是可移植的。如果我们的代码足够简单，那么整个创建过程将只涉及到几行命令；但如果我们的代码非常地多并且很复杂，那么创建过程也会相应地很大和很复杂。例如，创建使用 bzero 函数的这段代码，创建过程将需要知道在平台 P 是否可以得到函数 bzero，并且是否必须相应地设置 HASBZERO。复杂的代码可能还会有更多这种的依赖性。理想情况下，我们希望源代码和创建过程都具有可移植性。但遗憾的是，对于复杂的代码体，编写可移植的创建过程将会是非常困难的，甚至是不可能的；因为在不同的平台上，使用的是各种各样不同的创建工具。

在下面各节里，我们将讨论使编写可移植的 C++源代码和创建过程变得复杂的各种因素。

9.2　不断发展的语言定义

当写这本书的时候，C++还没有完全标准化；按计划，ANSI/ISO 标准要在 1996 年全部确定。在这之前，有另外一个实际的标准：*The Annotated C++ Reference Mannual* [ES90]，通常也称为 ARM。除了包含了广泛的注释和注解以解释这种语言的许许多多设计决定的基本原理，ARM 还包含了用于 ANSI/ISO 标准的基本文档。

第一个 C++翻译器（Cfront）源自于贝尔实验室，后来由 Novel 的 UNIX 系统小组对它进行进一步商业性开发并且推向市场，并正式命名为 C++语言系统。因此，在很长的一段时间里，Cfront 也被认为是一个实际的标准。

9.2.1　冲突

然而，ARM 和 Cfront 并不总是一致，某些已有的 C++实现和语言结构的解释有所分歧。有时候，这些实现和 ARM 之间的分歧不断引起很大的争论。考虑下面的代码：

```
class A {
private:
    class B { /*...*/ };
    B* f();
};
A::B* A::f(){ /*...*/ }      //合法吗?
```

ARM 肯定地说上面的定义是不合法的，因为私有类型 B 的引用是发生在它的包含类 A 的外部。然而，对于 Cfront 而言，上面的结构（语句）却是合法的——支持用户可以外联地定义 f 函数——但其他编译器却不允许这种结构。这个冲突将会由 ANSI/ISO 来解决。

注意，顺带提及一下，即使在 ANSI/ISO 标准完全公布之后，完全符合 ANSI/ISO 标准的代码也不能被保证肯定可以被移植。因为要让 C++实现（编译器）完全符合这个标准肯定需要相当长的时间——可能是好几年。

9.2.2 实现的完整性

现在许多 C++编译器并没有完全实现整个语言，也并不是所有的 C++编译器都实现了模板和异常处理，而这两者在 1990 年就已经提出来了。只有等到所有的（或大多数）C++的实现都符合 ANSI/ISO 标准，C++程序员编写的、符合 C++子集的代码，才可以被他们希望移植其上的平台所支持。当这本书出版的时候，并不是所有 C++编译器都完全支持这样的一系列 C++的新特性，这包括异常处理、运行时类型信息、成员函数返回类型的协变、名字空间、bool 类型、mutable 关键字、新的 cast 语法（static_cast, reinterpret_cast 和 const_cast）、模板的默认参数和模板内的模板参数。而且，某些 C++编译器还没有完全实现嵌套类：

```
class A {
public:
    class B;          //能实现吗?
    class C {
        void f();
    };
};
void A::C::f() { }  //能实现吗?
```

上面嵌套类 B 不完全的声明，和嵌套类 C 的函数 f 的定义，也并不为所有的 C++编译器所支持：某些编译器就算支持了这些结构，也可能不支持模板版本的这些特性（就是把上面的 A 改变为一个模板类之后的代码）。更严重的是，少数编译器甚至仍然不支持模板（这将导致许多可移植性问题，我们会在练习 9.7 看到这一点）。

当写这本书的时候，大多数 C++编译器的实现都是不完整的；因此，就有某些实现提供了对 C++的扩展（即不符合标准的特性）。例如大多数运行在个人计算机的 C++版本，为了使程序员可以操作系统中的某些进程，都会定义 near 和 far 关键字，但这种设计终究是不符合标准的。因此，使用扩展特性的程序员应该充分注意，他们正在阻止其代码失去将来可以移植到其他平台的可移植性。

9.3 不确定的行为

对于某些用法，ANSI/IS C++不会对 C++实现的行为强加任何要求。例如，对于左移或者右移运算符（<<或者>>），如果右操作数是负的，或者右操作数大于或者等于左操作数的字节数，那么结果将会是不确定的。在构造函数中，调用构造对象的纯虚函数也会导致不确定的行为，除非使用了显式限定。

在 C++语言定义中，大约有十几个结构具有未经定义的行为。根据 C++标准，编译系统

对这些包含有这些结构的程序的处理同时也是不确定的。例如，假设我们的程序是一个输入程序，那么编译器的处理行为将会是下面中的一种：

- 产生一个诊断信息；
- 中止；
- 删除文件系统所有的文件（虽然这种编译器早晚肯定会被淘汰）；
- 既不产生错误消息，也不产生警告消息，生成可执行代码。

而且，如果编译系统生成了可执行代码，那么这份代码的行为可能如下：

- 产生程序员所希望的结果；
- 产生不正确的结果；
- 中止；
- 删除文件系统中所有的文件。

遗憾的是，当程序发生不确定行为的时候，很少有C++实现会通知程序员。如果旨在编写可移植的C++代码，那么你必须具有小心谨慎的态度，以便避免编写这样的结构。

对于可移植代码而言，一个要特别关注的领域是内存布局（memory layout）——主要指：对象的大小和排列方式、指针和地址操纵。许多C++程序员都把可执行程序的地址空间看作是连续的字节序列，设置为适当值的char*或void*的指针变量都可以取到这些地址。然而，平台不是唯一的，也存在不符合这种简单内存布局模式的平台；因此，对于依赖于这种布局模式的代码，对这些平台而言，就是不可移植的。

9.3.1 排列方式和补全（padding[1]）

在C++中，许多类型都有排列方式限制，考虑下面的代码：

```
char x[sizeof(int)];
int* p = (int*) x;
*p = 0;
```

如果数组x并不能得到int类型的内存空间大小，那么第3条语句（*p = 0）可能会导致不确定的行为（程序可能会中止[2]）。

为了可以得到最大的可移植性，代码就不应该依赖于对象内部的布局，因为语言定义根本就没有保证这种布局不会发生改变。例如，下面的代码

```
int i;
char* p = (char*)&i;
```

并不能保证p肯定会指向i中的最低字节（或任何其他特定字节）。

现在考虑下面的代码：

```
static union {
```

[1] 译注：padding指在程序设计语言PL/1中，并置在串右边以把串扩展到所需长度的一个或多个字符或位。对于字符串，填充的是空格字符，对于位，填充的是零。

[2] 译注：实际上，在VC6上调试正常。sizeof(*p)\sizeof(*)的值都是4。

```
    int i;
    int bits: 32;
};
bit ^= 1;
```

即使我们知道上面的代码将会在 int 类型为 32 位的机器上运行，也不能保证赋值语句将会把 i 的最低位（或者任何其他特殊位）置位。

不仅如此，可移植代码还不能依赖于类对象的内部补全（padding）机制。例如，考虑下面代码：

```
struct X {
    char c1;
    char c2;
};
int main() {
    X x;
    char* p = &x.c1;
    ++p;
    //...
}
```

我们并不能保证当 p 执行自增之后，它就会指向 x.c2。

试图重叠不同的结构体也会导致许多可移植性问题。考虑下面的代码，它试图重叠结构体 X 和 Y：

```
struct X {
    char c1;
    char c2;
};
struct Y {
    int bits:8;
    char c2;
};
int main() {
    X x;
    Y* y = (Y*)&x;
    y->c2 = 0;
    //...
}
```

即使我们知道这段代码将会在 char 类型为 8 位的机器上运行，我们也不能保证 y->c2 赋值语句可以清除 x 中 c2 域的值[1]。

9.3.2 地址操纵

与大多数人的观点相反，C++语言定义并没有保证类型为 void*的变量可以指向程序地址

[1] 译注：在 VC6 中调试的结果为：不能改变。

空间的任意位置。特别地，指向函数的指针就不能可移植地转化为 void*：

```
void g();
int main() {
    void* q = (void*)g;      //不可移植的。
    //...
}
```

只有在那些 sizeof(void*)的大小足够容纳下函数指针的平台上，试图把函数指针转化为 void*类型的操作才有可能是成功的。因此，对于一个包含这类转型的程序库，在某些平台或许可以正确地编译和执行，在另一些平台可能就会出现奇怪的行为，在其他一些平台也可能是不能编译的。

另外，某些程序员试图使用 long 类型的变量来存储和操纵内存地址，但这种做法也是不可移植的——long 类型的大小也可能容纳不下一个指针。譬如，当下面的代码执行后：

```
char x[10];
long addr = (long)x;
char* p = (char*) addr;
```

p 并不一定指向 x[0]。实际上，在大多数平台[1]中，p 都不会指向 x[0]。

9.4 合法但不可移植的代码

C++继承了 C 语言的许多实现性定义（implementation-defined）的行为和未经指定的（unspecified）行为，并增加了一些新的行为。虽然 C++程序可以合法地依赖于实现性定义行为和未经指定行为，但这么做将会导致这部分程序代码不能获得最大化的可移植性。遗憾的是，我们却经常会不自觉地编写具有这些依赖性的代码。

9.4.1 实现性定义的行为

C++语言拥有 40 多个实现性定义结构。例如，对于右移运算符（>>），当它是确定的(见 9.3 节)，并且第一个操作数是有符号类型的负值时，这个操作的结果就是实现定义行为的。这个操作可能是一个逻辑右移操作（最高位用 0 来填充），也可能是一个有符号的右移操作。对于程序中每一个实现性定义结构，C++编译器实现都需要指定它实际要实现的特定行为。因此，由于 C++编译器的差异，包含这类结构（指上面的实现性定义结构）的程序的行为往往就有所差异。

譬如，char 类型究竟是有符号类型的还是无符号类型就是实现性定义的，考虑下面的代码：

```
char c;
//...
c >>= 1;
```

[1] 译注：在 VC6 中和 BC 中，P 都指向 x[0]。作者这里“大多数平台”指 Unix 平台。

如果 char 是无符号类型的，或者 operator>>被定义为逻辑右移运算符，那么上面代码将会清除 c 的符号位；如果 char 是有符号类型，并且 operator>>被定义为算术右移运算符，那么 c 的符号位会继续存在（即不会被清除）。

ARM 里已经给出了完整的实现性定义行为的集合，ANSI/ISO 也给出了这个集合。遗憾的是，当程序的构建依赖于这些实现性定义行为的时候，C++编译器并不会告诉程序员这一潜在的陷阱。因此，这幅沉重的担子自然就落在希望编写可移植代码的程序员身上，程序员必须以非常小心谨慎的态度避免编写含有这类结构的代码。

9.4.2 未经指定的行为

C++留下的程序结构的行为中至少有 6 个是未经指定的，然而，存在这类未经定义行为的 C++实现却会执行任何可能的操作；于是，ANSI/ISO C++标准给出了当结构的行为是未经指定时可能执行的行为列表。另外，C++实现也没有必要用文档说明当在程序中碰到这类未经指定行为时会执行哪种操作。

例如，函数参数的求解顺序就是未经指定的。因此，考虑下面给定的 print_args 函数的定义：

```
void print_args(int arg1, int arg2){
    cout << arg1 << arg2 << endl;
    return;
}
```

如果我们如下调用 print_args：

```
int i = 0;
print_args(++i, i);
```

那么输出结果可能是：

```
1   0
```

也可能是：

```
1   1
```

而 C++编译器并不需要说明它将产生哪个结果；不仅如此，函数调用不同，同一个实现的行为也可能是不同的。

定义在不同翻译单元的全局对象的初始化顺序也是未经指定的。C++语言未能指定这个顺序往往会给程序库编写者带来许多困难。12.1 节将会更详尽地讨论这个问题，以及程序库作者可以为初始化执行什么样的操作。

依赖于未经指定的行为的程序在被构建和执行的时候偶尔也会照预期地运行。例如，假设我们的程序包含了两个翻译单元 f 和 g，并且还包含了一个初始化依赖性，即只有当定义在 f 的全局变量的初始化时间比定义在 g 的全局变量的初始化时间早，程序才能准确无误地运行。那么假设我们如下链接这个程序：

```
% link f.obj g.obj
```

如果我们的编译系统拥有这样的特性：翻译单元中全局变量的初始化顺序和链接命令中指定

的翻译单元的顺序一样，那么当用这个编译系统创建程序时，我们的程序就可以准确无误地运行。然而，当我们要把我们的程序移植到具有不同编译系统的平台，并且在这个平台运行这个程序的时候，程序的行为就在所难料了。

因此，对于那些旨在编写可移植代码的C++程序员，要有小心谨慎的态度，才能避免依赖于这类未经指定的行为。

9.5 实现依赖性

当然，C++语言定义并没有陈述任何关于语言应该如何实现的问题（ARM确实给出了许多广泛用于实现的技术，但所有这些技术都不是强制要求的）。因此，那些旨在编写可移植代码的程序员应该避免依赖于某个特殊的C++实现的细节。

为了理解程序员为何会热衷于依赖语言的实现细节，我们考虑下面这个例子：

```
f.c
    void f(int i){
        //...
    }
    void f(char c) {
        //...
    }
main.c:
    void f(int);
    void f(char);
    int main() {
        f(0);
        f('0');
        //...
    }
```

当链接这个程序的时候，main.c中每个f函数的引用都会调用定义在f.c中相应的实例。为了确保实际情况确实如此，现今所有C++编译器都在目标代码中，把参数类型编码（代码化）到函数的名称中。例如，C++实现可能会使用编码后的名字f__Fi和f__Fc分别代表f(int)和f(char)。编码化后的函数名称也会随着编译系统的不同而各不相同，甚至由于系统版本的差异也会带来一些差异。因此，对于那些依赖于这类名称编码机制的程序员，编写出来的代码将是不可移植代码。

然而，假设我们的编译器会由于创建两个f函数的定义，而变得异常低效；并且假设我们知道如何在其他的语言（如C）编写更加高效的代码；那么我们可能会用下面的C代码来取代f.c中的定义：

```
/* C代码 */
void f__Fi(int i){
    /* 比较高效的C代码 */
}
```

```
void f__Fc(char c){
    //* 比较高效的C代码 */
}
```

以这种方式重写这段代码将会很快，也很简单；但遗憾的是，这段代码仍然是不可移植的。为了既得到效率又具有可移植性，我们首先要改变C函数的名称（见练习9.5），使用如下命名：

```
/* C代码 */
void f_int(int i) { /*...*/ }
void f_char(char c) { /*...*/ }
```

接下来，我们改写f的C++定义，如下所示：

```
extern "C" {
    void f_int(int i);
    void f_char(char c);
}
void f(int i){ f_int(i); }
void f(char c){ f_char(c); }
```

从上面可以看到，和前面一样，编写可移植代码需要花费更多的精力。

9.6 可移植的数据文件

通常，我们都希望数据文件可以是跨平台可移植的。为了能够得到这种可移植性，数据文件应该以与写入时相同的方式来读取。例如，考虑下面的结构体：

```
struct X{
    char c1;
    char c2;
};
```

假设我们如下把一个X类型的值写入一个文件：

```
X x;
ofstream f("datafile");
f.write((const char*)&x, sizeof(x));
```

接下来，我们使用相应的读取函数，并使用相同的类型来读取这个值：

```
X x;
ifstream f("datafile");
f.read((char*)&x, sizeof(x));
```

这个实现方法保证了：如果可以符合下面的条件，那么读操作就可以读取被写入的值。

- 使用同一个编译系统编译包含读操作和写操作的程序，或者编译器将会产生具有相同排列方式和补全方式的目标文件。
- 在同一个平台执行读操作和写操作。
- 读操作和写操作都可以成功运行。

然而，这个实现方法并没有保证在一个平台编写的文件一定可以在另外的平台正确地被

读取。如果在两个平台中，X的外部描述是不同的，那么读操作将不能得到正确的写入值。

一种实现数据文件跨平台可移植的方法是，按照预先定义的顺序，每次都只读取或者写入一个字节。例如，如果我们如下写入X对象：

```
X x;
ofstream f("datafile");
f.write(&x.c1, 1);
f.write(&x.c2, 1);
```

我们如下读取X:

```
X x;
ifstream f("datafile");
f.read(&x.c1, 1);
f.read(&x.c2, 1);
```

那么对于每个和平台P具有相同char类型值的外部描述的平台，都可以移植在平台P写入的数据文件。当然，如果X包含了任何非char类型的数据成员，那么我们就必须递归地、每次一个字节地、按照原先预定顺序写入或者读取这个数据成员（见练习9.2到练习9.4）。

然而，对可移植文件而言，这种一次一个字节的技术却会影响运行时间（即效率）。显然，下面代码

```
f.write((const char*)&x, sizeof(x));
```

可能会比这些代码具有更快的效率：

```
f.write((const char*)&x.c1, 1);
f.write((const char*)&x.c2, 1);
```

实际上，一次一个字节写入和读取的额外开销是很小的，通常不会给这种技术的应用带来很大的障碍。

9.7 模板实例化

用户平台上的模板实例化机制会影响C++程序库的设计与实现。当程序库在它的实现或者接口中包含模板的时候，某些实例化机制要求源代码以某种特定的方式组织。如果我们要预先实例化任何模板（见4.2.2小节），那么程序库的创建过程就必须知道如何使用实例化器。但遗憾的是，实例化器往往是各不相同的。因此，编写跨实例化器可移植的代码是相当困难的，或者是不可能实现的。旨在重用C++代码的设计者必须清楚了解，他们的潜在用户为了创建和使用他们的程序库，而需要调用的各种模板实例化机制。在下面的几节里，我们将讨论各种模板实例化机制。

9.7.1 自动的实例化器

C++的ANSI/ISO定义要求C++开发环境能够自动实例化被创建程序使用的所有模板特化（参照前言中特化的定义）。假设我们的程序库提供了下面两个模板：

```
List.h:
    template<class T>
    class List {
    public:
        void insert(const T& t);
        //...
    };
Hashtable.h:
    template<class T>
    class Hashtable {
    public:
        Hashtable();
        void insert(const T& t);
    private:
        List<T> the_table[1024];
        //...
    };
```

一个List<T>是一个元素类型为T的链表。Hashtable<T>是一个固定大小的、元素类型为T的哈希表，这个哈希表使用了开放链（就是说，当有多于一个的值散列到同一个位置时，就把这些值都保存在以这个位置为头节点的链表中）。Hashtable的实现还使用了一个固定大小的List数组，即the_table。

假设我们程序库的用户创建了一个Hashtable<Widget>，其中Widget是用户定义的一个类；那么自动的实例化器必须能够检测到包含程序使用了Hashtable<Widget>和List<Widget>；这个实例化器并且还能够实例化这两个类所需要的所有对象。

有几种方法可以用来设计自动的实例化器，下面两个小块将分别讨论两种常用的设计方案（另一种设计方案在练习9.9讨论）。

1、编译期实例化器

在每个翻译单元里，编译期实例化器将会直接或者间接地实例化这个翻译单元使用的所有模板特化。编译期实例化器为了可以顺利执行实例化，还会要求用户#include所有需要的头文件（这一点与链接期实例化器有区别，我们将在下一小节讨论链接期实例化器）

假设Hashtable::insert调用了List::insert：

```
template<class T>
void Hashtable<T>::insert(const T& t) {
    int i;
    //...
    the_table[i].insert(t);      //调用List::insert
    //...
}
```

如果用户的翻译单元调用了Hashtable<Widget>::insert，那么编译期的模板instantiator（实例化器）将会在这个翻译单元的目标代码中产生：Hashtable<Widget>::insert的定义、List<Widget>::insert的定义和其他任何这两个函数直接或者间接使用的特化。

当然，在每个翻译单元的目标文件中，产生所有这个翻译单元直接或者间接使用的特化，将会导致一个这样的结果：在几个需要链接成一个程序的翻译单元目标文件中，将会有重复的定义存在。当这个程序被链接的时候，只有一份重复定义的拷贝会被合并到可执行程序中（即其他的重复定义将不起作用）。因此，编译期实例化器往往和能够去除重复定义的特殊链接器配合使用。

编译期实例化器最大的缺点就是效率不高。如果在n个翻译单元中都存在某个相同的特化，那么就需要编译n次，并产生n份拷贝。对于那些经常使用很多模板的大型程序开发者而言，这些重复的时间和空间将会是浪费的，所带来的感觉也只能是令人沮丧的。

2．链接期实例化器

链接期实例化器将在链接期产生程序需要的所有模板特化。当一个程序被链接时，这个实例化器将会由于缺少特化而不断重复创建目标代码，直到所有的特化都已经完全。生成缺少的特化需要不断地运行编译器，因此，链接期实例化器将在链接期编译代码（这是和编译期实例化器的区别）。编译后，生成的目标代码被放在一个用来存储模板特化的位置，通常称为储存库（repository）。

这种实现的算法从概念上看来是非常简单的，然而，如果要在程序首次编译的时候，程序能高效正确地运行，并且在程序由于改变而重新编译时，程序也可以高效地运行，那么这个实现就很困难了。幸运的是，编写一个正确而且高效的模板实例化器是编译器开发商所关注的问题，并不是可重用代码设计者和实现者关心的对象。

为了简化实例化，某些链接期实例化器要求模板源代码的组织形式符合某种约定。假设Hashtable的定义位于头文件Hashtable.h中：

```
Hashtable.h:
    template<class T>
    class Hashtable {
        //...
    };
```

那么某些实例化器会要求Hashtable中所有的非内联成员函数和静态成员变量的定义都应该放在一个名为Hashtable.c的文件中，这个文件和Hashtable.h位于同一个目录下：

```
Hashtable.c:
    template<class T>
    void Hashtable<T>::insert(const T& t) {
        //...
    }
    //...
```

链接期实例化器所要求的源代码组织形式通常都是比较易于实现的，但这却会带来移植性问题。因为不同的实例化器将会施加不同的要求，而且有时候这些要求又是互相冲突的。因此，编写对于具有不同实例化器的平台仍然是可移植的代码，就需要在代码的创建过程中花费额外的精力。

3．两种自动实例化器效率之比较

典型的链接期实例化器只会在一个位置（即 repository，储存库）产生模板特化后的代码，而前面的编译期实例化器是在每个直接或者间接使用特化的翻译单元都产生这些代码。因此，与编译期实例化器相比，链接期实例化器的输出占用空间更少。

一个幼稚的分析可能会得到这样的结论：链接期实例化器要比编译期实例化器快。实际上，与编译期实例化相比，在链接期实例化可能更快、可能是一样快，也可能更慢。考虑下面的代码：

```
main.c:
    #include<Hashtable.h>
    #include<Hashtable.c>   //用于编译期 instantiator
    int main() {
        Hashtable<int> table;
        table.insert(7);
        return 0;
    }
```

让我们假设函数 Hashtable<int>::Hashtable()和函数 Hashtable<int>::insert(const int&)并没有使用任何其他的模板特化。如果我们使用编译期实例化器来创建这个程序，那么这两个函数的定义将会在 main.c 的目标代码生成中；当这个目标代码被链接时，所有需要的特化都已经给出了，因此也就没有任何重复的定义需要识别和删除了。

现在假设我们使用链接期实例化器来创建这个程序。当 main.c 被编译的时候，并不会生成任何特化；当链接 main.c 的目标代码时，链接期实例化器就需要创建 Hashtable<int>::Hashtable()的定义和 Hashtable<T>::insert(const int&)的定义，为此，这个实例化器可能创建一个如下的文件：

```
Hashtable_int.c:
    #include<Hashtable.h>
    #include<Hashtable.c>
    #pragma instantiate Hashtable<int>::Hashtable()
    #pragma instantiate Hashtable<int>::insert(const int&)
```

完成了这个文件之后，链接期实例化器就会调用编译器编译这个文件。当编译器返回时，这个实例化器再把编译后的文件添加到需要链接的程序中去。

当我们使用编译期实例化器创建这个程序时，Hashtable.h 和 Hashtable.c 的代码只会被编译一次——发生在 mani.c 被编译的时候；而当我们使用链接期实例化器来创建这个程序时，Hashtable.c 和 Hashtable.c 的内容都被编译了两次——也是发生在 main.c 被编译的时候。因此，对于这个程序，链接期实例化器就可能要比编译期实例化器慢；对于一个大的程序，链接期实例化器可能会在不断编译头文件上面花费很多时间。

模板实例化的效率——无论是使用链接期实例化器还是使用编译期实例化器——至今仍然是 C++用户关心的焦点。在 4.2.2 和 4.2.3 小节里，我们给出了程序库用户提高模板实例化效率的一些技术。

9.7.2 人工实例化

因为实现一个正确的、高效的、自动的实例化器是很困难的，所以某些（不完全）C++开发环境并没有提供实例化器。这样的开发环境就强制用户必须自己通过各种人工实例化机制，来指定实例化的内容和位置。

ANSI/ISO C++可能会提供人工实例化指令：

```
template class Hashtable<Widget>;
```

这条语句的意思是，对于Hashtable<Widget>所有的非内联成员函数和静态数据成员，在这里产生它们的定义。函数模板的语法是相似的；例如，假设下面的模板

```
template<class T> void f(const T&);
```

那么语句

```
template void f(const Widget&)
```

的意思就是在这里产生f<Widget>的定义。

现今某些C++开发环境能够辨识的人工实例化结构和上面描述的是不同的。另一种广泛使用的结构使用#pragma，这个用于指定实例化内容的语法可以让程序员表达实例化的内容，例如“在此产生Hashtable<Widget>中名称以‘put’开头的所有非内联函数的定义”。当然，依赖于这些机制的代码是不可移植的。

某些C++实现提供了自动和人工两种实例化机制。明智地使用人工实例化可以减少由于单一使用自动实例化而导致的创建（build）时间。程序员通常都可以判断出最佳的实例化内容和位置，而自动的实例化器却不能。如我们在9.7.1小节讨论的那样，对于大型的复杂程序，自动化实例化器往往会浪费大量的空间和时间；而人工实例化可以减少这些资源消耗，美中不足的是，这些节省是以程序员必须花费时间决定实例化的内容和位置为代价的。

1．使用人工实例化

由于传递的相关性，所以决定实例化的内容和位置是很困难的。假设我们的程序使用了一个Hashtable<Widget>，并且我们决定人工实例化我们程序中使用的所有特化：

```
template class Hashtable<Widget>;
```

我们还必须记住要人工实例化那些Hashtable<Widget>所需要的特化，如：

```
template class List<Widget>;
```

如果List<Widget>使用了其他的模板，我们还需要实例化这些模板。

如果我们是Hashtable的实现者，那么实例化List<Widget>将不是费力的事情；然而，如果我们根本就不了解Hashtable的实现细节，那么我们可能就只能写出上面第一个实例化，当链接程序的时候，将会有错误告诉我们：和List<Widget>相关的某些函数和成员没有定义（即没有实例化）：

```
error: undefined function:
List<Widget>::insert(const Widget&)
```

如果程序非常地大，我们就很难知道究竟何处需要List<Widget>::insert的定义；况且我们可能不愿意花费时间来找出这个位置。于是，我们会增加上面第二个人工实例化到我们的

程序中。因此，我们就必须不断重复程序的链接过程，不断手动地增加实例化，直到所有的函数和成员都有定义为止。

当增加一个人工实例化的时候，还必须#include 相应的头文件：

```
#include<List.h>
#include<Widget.h>
//...
template class List<Widget>;
```

然而，错误消息并没有告诉我们包含 List 声明的头文件名称。因此，为了使用户确定这个文件的名称，包含 Hashtable 和 List 的程序库要么使用统一的文件命名约定，要么在用户需要人工实例化某个模板时，对该模板所需要包含的头文件，程序库给出相应的文档。虽然 List 的存在只是程序库的实现细节，并且也并不允许外部公共使用，但在这里，我们不得不公布 List 模板定义的位置。

链接一个大的程序通常是一个很慢的过程。为了辨识所有需要的人工实例化，而不断重复地链接程序是一个更慢的过程。为了减小需要链接的次数，程序库的文档可以指定实例化的相关性：

实例化 Hashtable<T>的程序必须同时实例化 List<T>。

当程序员读到这个文档的这些内容的时候，将会一次人工实例化这两部分，从而节省了一次链接过程。

2．人工实例化和代码更改

如果程序就只编写一次，并且从不更改，那么人工实例化就是很实用的技术；然而，当我们更改程序时，人工实例化却会遇到重重困难。假设我们重新实现 Hashtable，让它采用封闭链而非开放链（就是说，当冲突出现的时候，通过某种算法，我们在表中找到另一个位置来存储这个给定的值），那么我们需要如下修改 Hashtable：

```
tomplate<class T>
class Hashtable {
private:
    T* table[1024]; //上一版本这里是 List<T> the_table[1024].
    //...
};
```

所定表的每个入口(即指针)都指向存储在位置的值，如果这个值不存在，那么这个指针为 0。因为我们不使用开放链，所以这里将不再需要使用 List 来实现 Hashtable。

当完成了这个改变之后，使用 Hashtable<Widget>的程序可能仍然需要（也可能不需要）实例化 List<Widget>（因为在程序的别处也可能使用了 List<Widget>）。如果程序不再需要 List<Widget>，那么包含下面人工实例化指令

```
template class List<Widget>
```

将会导致产生不需要的代码，增加了可执行代码。

即使一个很小的改变，也可能会导致程序需要的实例化发生很大的改变。那么，在发生了改变之后，程序员应该如何决定所需要改变的最小实例化集合呢？如果使用的是人工实例

化，那么一种肯定可以找到不需要的实例化集合的现实方法就是，删除所有的人工实例化，从头开始重复整个人工实例化过程。

9.8 运行期程序库

多数平台都提供了C++程序员可以使用的程序库集合。例如，具有文件系统的平台就会提供用于创建文件和修改文件的函数；而不含文件系统的平台——例如，一个用于有汽车发动机内部执行嵌入式程序的平台——将不提供这些函数。因为不同平台提供的程序库差异往往非常大，所以C++代码应该只使用那些所有平台共有的功能，才能使这部分代码是可移植的。

虽然ARM中并没有提到，一个标准的程序库应该是根据ANSI/ISO C++标准定义的；但当这本书编写的时候，ANSI/ISO标准草案中声明一个标准的C++程序库应该具有下面领域的功能：

- 语言支持；
- 诊断；
- 一般实用程序；
- 局部化；
- 容器；
- 迭代器；
- 算法；
- 数值类型；
- 输入和输出（即iostream程序库）。

直到今天，大多数C++实现仍然没有提供整个草案程序库，而只是提供下面的功能[1]：

- iostream程序库的某些变型。这些实现与草案标准中的iostream实现是不同的；
- 大多数ANSI/ISO C（不是C++）程序库，经过了某些修改。

实际标准的iostream库是贝尔实验室在1988年创建的。这个程序库有几点优于以前的stream程序库。首先，早期的stream库和后来的iostream库，都由实际标准编译器cfront所支持；于是ANSI/ISO C++委员会就抓住这个机会来改善iostream库的设计；当编写这本书时，iostream库的接口不断地发生改变，甚至几乎没有C++实现可以完全支持草案标准的iostream库。在大多数C++实现都提供ANSI/ISO的标准iostream库之前，那些旨在编写可移植代码的程序员，在编写含有输入操作和输出操作的代码时候，都必须以非常小心谨慎的态度来编写对所有的iostream程序库都适用的代码。

ANSI/ISO C程序库，经过某些修改，会演化成ANSI/ISO C++程序库的一个子集。之所

[1] 译注：这只适用于作者写作本书的1995年左右。

以需要经过某些修改，是因为C和C++的类型系统有某些小的差异。例如，考虑在C程序库声明的函数strchr：

```
char* strchr(const char* p, int c);      /* ANSI/ISO C */
```

这个函数返回一个指针，指向非null结尾的字符串p中第一次出现字符c的位置；如果c不在字符串中，那么将返回0。第2个参数的类型为int，这是因为在C中，文字字符（如x）的类型都是int，而不是char；文字字符在C++中才使用字符类型char。只要仔细观察，不难发现C的strchr函数包含了一个类型漏洞：

```
/* C 代码*/
void modify_string(const char* p) {
    char* q = strchr(p, p[0]);
    q[0] = 'x';
}
```

在这里，我们可以使用strchr的返回值来改变p指向的字符串（strchr函数中p的类型是const char*的）。为了解决这个问题，在ANSI/ISO C++中，我们将如下重载和声明strchr函数：

```
//C++ 代码
char* strchr(char* p, char c);
const char* strchr(const char* p, char c);
```

当编写这本书时，ANSI/ISO C++委员会正在讨论应该如何把这个更改应用到用于C++的C程序库中。

即使在ANSI/ISO C++标准程序库得到官方定义之后，也并不是所有的平台都会提供整个程序库。考虑一个用于在汽车发动机中执行的程序，当需要执行下面语句时，这个程序应该怎么做呢？

```
cout<<" Hello, Officer!\n "
```

嵌入式程序通常都没有用于打印消息的相应输出设备；因此，一个用于给程序编写嵌入式代码的平台，可能不会提供整个iostream程序库，或许只在测试模式提供iostream程序库（见5.3.1小节）。

大多数平台还提供了标准C++程序库以外的程序库。例如，许多平台提供了定义在System V Interface Defination(SVID)[AT&89]的程序库。SVID定义了一个用于访问很多功能的C接口，C++程序员也可以访问这个接口。另外，Portable Operationg System Environment for Computer Enviroments Standard(POSIX)[IEE90]也定义了一个函数库，主要用于和底层操作系统进行交互。某些平台还提供了标准文档没有描述、但却是实际标准的程序库。我们将不在这里讨论这些程序库。C++程序员如果希望把代码移植到新的平台，就应该充分了解这个平台所提供的程序库。

9.9 其他移植性问题

平台之间的许多差异使C++源代码和创建过程（build procedure）的移植性变得更加复杂。

在这里，一个还有问题的方面就是系统命令——例如，用于显示文件内容的命令，或者用于创建程序的命令（通常称为 make 或者 build）。大多数平台都提供了一个或者多个命令行解释器，或者称为 shell，在 Shell 中就可以执行系统命令；而且，几乎每个 shell 都提供了它本身的编程语言。遗憾的是，平台的差异也使 shell 和系统命令的差异非常显著。即使相同名称的命令，但在不同的平台上，也可能做不同的事情，或者拥有不同的接口。系统命令之间的差异令可移植地使用标准 C++库的函数系统的可能性几乎为 0（函数系统传递一个字符串参数给操作系统，然后这个字符串被作为命令解释并执行）。在具有这些不一致性的平台之间，试图编写可移植的创建过程是很具挑战性的。

另一个源代码的可移植问题就是文件系统。大多平台都提供了一个层次化的文件系统。遗憾的是，文件系统的命名文件的语法多种多样。例如，在 Unix 系统下，为了在包含当前目录的 dir 目录下引用 file1.c 文件，我们可以使用：

```
../dir/file1.c
```

然而在 VMS 系统下，我们应该使用：

```
[-.dir]file1.c
```

而在 Windows 系统下，我们就必须这样使用：

```
..\dir\file.c
```

这些区别影响了 C++源代码和创建过程的可移植性，因为它们都要经常操作路径名称。例如，下面的代码

```
#include"../dir/file1.h"
//...
ifstream f("../dir/file2.c");
```

可能只对那些辨识 UNIX 系统路径语法的平台之间才是可移植的。为了提高代码的可移植性，某些平台可以辨识几种路径语法。例如，大多数（几乎所有）运行在 Windows 平台下的编译器都可以辨识 Windows 和 UNIX 的路径语法。

我们很少提及另一个可移植问题的来源：窗口系统。不同的平台会提供不同的窗口系统，并使用不同的应用程序编程接口（API）。试图编写对几个窗口系统都可以移植的代码也是非常困难的。某些程序库通过在几个窗口系统 API 上放置一个单一接口，来解决这个窗口系统问题。

9.10　总结

现今，编写高质量的可移植 C++代码是很具挑战性的；挑战部分来自于 C++语言的不断发展。在 C++实现应该如何解释某些特定的结构方面，还存在着一些争论。而且，语言的某些实现还不够完整。

即使在 ANSI/ISO C++标准最后定案之后，C++语言所允许的合法程序也未必是可移植的。C++还继承了 C 的许多不确定的行为、未经指定的行为和实现定义性行为，并且增加了

一些新的行为。特别地，为了编写可移植代码，我们应该特别关注内存和对象布局。

在不同的C++编译器实现中，模板实例化机制也多种多样。某些自动实例化机制要求模板代码是以特定的方式组织的，但不同实现的要求也会随着实现的不同有所不同。人工实例化机制使用了多种指令，从而使用户可以控制模板实例化。因此，使模板代码的使用具有可移植性将需要更多的努力。

最后，如后一些因素也会使具有可移植代码的程序变得更加复杂：标准、非标准运行期程序库、系统命令、文件系统和窗体系统。

9.11 练习

9.1 详细讨论下面代码段的可移植性：

```
Exercise9.1a.c:
    namespace{
        class Secret {
            char _c;
            / ...
        };
    };
Exercise9.1b.c:
    ofstream o("dir1/file");
    o << "Engage!\n";
Exercise9.1c.c:
    class Oops { /*...*/ };
    void shifty() throw(Oops) {
        char c = -1;
        try {
            c <<= 4;
            //...
        }
        catch (Oops) {
            //...
            throw;
        }
    }
Exercise9.1d.c:
    #include<iostream.h>
    static union {
        char c1, c2;
        //...
    };
    main() {
        c1 = 'a';
        if(c2 == 'a')
```

```
        cout << "as i expected\n";
    }
```

9.2　9.6 节给出了如何可移植地写入和读取 char 序列的方法

a. 编写一对函数 portablewrite(ostream& o, long l)和 portableread(istream& i, long& l)，分别以预先定义的顺序写入和读取 long 类型的值，并且每次只操作（即读取或写入）一个字节。

b. 指定在什么样的条件下，在平台 P 用 portablewrite 函数写入的 long 类型值才能被平台 Q 的 portableread 函数读取。

c.（*）编写一对函数 portablewrite(ostream& o, double d)和 portableread(istream& i, double& d)，分别以预先定义的顺序写入和读取 double 类型的值，并且每次只操作一个字节。

d.（*）指定在什么条件下，在平台 P 用 portablewrite 函数写入的 double 类型的值才能被平台 Q 的 portableread 函数读取。

9.3　（*）假设我们写入一个类型为 T 的值 val1 到 ostream o 中，其中 o 指向文件的开始位置，代码如下：

```
T val1;
o << val;
```

我们接着把这个结果文件拷贝到别的平台，然后我们运行其他的程序，在这个程序中，用一个类型为 T 的值 val2 来从 istream i 中读取数据，其中 i 指向文件的开始位置：

```
T val2;
i >> val2;
```

a. 准确地说，如果 T 为类型 char，那么在什么条件下，val2 才等于 val1 呢？

b. 如果 T 为类型 int，那么在什么条件下，val2 才等于 val1 呢？

c. 如果 T 为类型 float，那么在什么条件下，val2 才等于 val1 呢？

9.4　（*）再次假设我们写入一个值 val1 到 ostream o 中，其中 o 指向文件的开始位置，这一次使用 write 函数：

```
T val1;
o.write((char*)& val1, sizeof(val1));
```

然后我们拷贝这个结果文件到一个不同的平台，接下来运行这个平台的其他某个程序，在这个程序中，使用 read 函数从流中读取值，并把值存储在 val2 中：

```
T val2;
i.read((char*)&val2, sizeof(val2));
```

a. 准确地说，如果 T 为类型 char，那么在什么条件下，val2 才等于 val1 呢？

b. 如果 T 为类型 int，那么在什么条件下，val2 才等于 val1 呢？

c. 如果 T 为类型 float，那么在什么条件下，val2 才等于 val1 呢？

9.5　我们为什么说（在 9.5 节）为了编写一个可移植的程序，我们不得不改变 C 函数 f__Fi 和 f__Fc 的名字呢？

9.6　假设模板实例化器 T1 要求模板函数的定义被包含在一个和这个函数具有相同名字的文件里；另外，如果文件里还包含有函数的声明时，在函数名后面添加一个不同的后缀给

这个声明命名。进一步假设模板实例化器 T2 要求函数模板的定义将会在所有的需要调用这个函数的翻译单元存在。

a. 要如何组织程序库的源代码，才能使这段代码对运行 T1 的平台和运行 T2 的平台都是可移植的？

b. 在 a 部分的解决方案是否可以最小化用户在两个平台的编译时间？

9.7 （*）如果我们编写的代码要对两个平台都是可移植的（使用自动实例化器的平台和只支持人工实例化的平台），那么将会导致什么问题呢？

9.8 （**）在 4.2.2 小节我们提过程序库应该预实例化某些的模板特化。

a. 编写一个具有如后功能的程序：这个程序将把给定特化的集合添加到目标代码档案文件中。在你的解决方案中，可以任意使用希望的平台和实例化器。

b. 你的程序可移植吗？

9.9 （**）在 9.7.1 小节中，我们讨论了自动实例化器的两种常用的设计。另外一种常用的设计是，当编译翻译单元 t 时，把 t 直接或者间接使用的、还没在储存库的特化都实例化到储存库中。讨论这种设计的优点和缺点。

9.12 参考文献和相关资料

Lipin[Lap87]讨论了编写可移植的 C 源代码和创建过程，这部分内容对 C++程序员也是很有用的。对于旨在 UNIX 平台编写可移植代码的程序员，Stevens 的作品[Ste92]将是必不可少的阅读资料。

不确定的和未经指定的行为的危害性已经深为编程开发团体所知，在现代的一些编程语言（如 ML[Wik87]）中，可执行程序不能含有不确定的和未经指定的行为。

Harbison 与 Steele[HS91]的第 6 章讨论了内存和对象的布局。

可移植地写入和读取浮点数的值是困难的。有兴趣的读者可以看 Clinger[Cli90]的论文和 Steele 与 White[SW90]。

McCluskey 与 Murray[MM92]讨论了 C++模板实例化的困难。Novell 的 C++ Language System Release 3.1 Manual, Selected Readings [Nov92]描述了自动链接期实例化器的实现。

Vilot[Vil95]讨论了 C++标准程序库。Plauger[Pla95]详细讨论了程序库的草稿版本（定于 1994 年 2 月）。Teale[Tea93]讨论了各种 iostream 实现的区别。C 标准程序库定义在 ANSI/ISO C 标准[ANS89]中。标准模板程序库是刚新增到 ANSI/ISO C++草案标准中的——内容太多了以致未能在这本书中讨论；Vilot[Vil94]、Stroustrup[Str94b]和 Plauger[Plag95]都提供了更多的信息。

练习 9.1 是受到 Becker 关于移植性的讨论[Bec94]的激发而编写的。

第10章

使用其他程序库

如果有一个更好的系统可以使用，就引入并使用这个系统，否则就继续使用我的这个系统。
——Horance

在这一章里，我们将讨论C++程序库是否应该重用其他程序库的某一部分。我们还将讨论重用其他程序库代码的缺点：需要用户得到可重用的代码，需要充分考虑重用后的效率，还要关注由重用其他程序库所带来的潜在的名字空间冲突，以及要考虑与可重用程序库保持版本同步问题。我们还讨论了编写自含式程序库——用来取代重用其他程序库的一种方法。

10.1 为何要重用其他程序库

有时，当编写一个程序库的时候，我们会发现自己往往想要重用其他程序库的部分代码。例如，假设我们在编写一个程序库MEDLIB，它主要用于医学应用程序。我们希望程序库可以提供许多医学应用程序特定领域的功能——就是指这个程序可以用于医院、医学实验室和其他医疗机构。因为许多医疗应用程序都需要使用诊断测试——验血、X光透视、CAT扫描等等，所以我们在自己程序库里提供了一个用于描述诊断测试的类Diagnostic。对于每个给定的诊断测试集合，我们还提供了一个函数，用于给应该为诊断付费的人开出诊断发票。

```
Void send_invoices(const Some_type<Diagnostic>& diags) {
    //给每个诊断清单中的诊断都开发票。
}
```

那么对于这个函数的参数，我们应该使用什么集合类型（即上面的some_type）呢？如果我们的程序库现在没有集合类，那么有两种选择来实现上面的some_type：可以设计和实现自己的集合类，也可以重用其他程序库已经存在的集合类。

对于上面的两种选择，我们有充分的理由选择后者。首先，如果已经存在一个合适的集

合类，我们又何苦再重新写一个呢？其次，我们或许只是医疗程序库的热心设计者，而不是容器类程序库的设计者，可能没有编写高质量容器程序库的经验。再次，如果我们是可重用程序库的提供者，当然愿意重用自己编写的程序库了。

假设我们选择了重用，并且在程序库 CONTAINER 中发现了一个可以重用的类 Set：

CONTAINER 中的 Set.h：

```
template<class T>
class Set {
    //...
};
```

我们如下完成开发票函数的接口：

```
Medical.h:
    #include<Set.h>
    void send_invoices(const Set<Diagnostic>& diags);
    //...
```

我们的用户也将会如下编写代码：

```
    #include<Medical.h>
    int main() {
        Set<Diagnostic> diags;
        //...
        send_invoices(diags);
        //...
    }
```

从上面可以看出，重用其他程序库的代码产生了一个不但优雅而且实现简单的接口。

10.2 使用其他程序库的缺点

遗憾的是，在可重用代码中使用其他程序库通常都会带来许多缺点。使用其他程序库将强制要求用户得到可重用程序库，还可能会损害效率，或者引入潜在的名字空间冲突，并且还需要和可重用程序库的版本更新保持同步。

10.2.1 获得可重用程序库

重用其他程序库的第一个缺点是：用户必须得到被重用的程序库。在我们上面的例子里，对于那些使用 MEDLIB 库的程序员，他们必须能够得到 CONTAINER 库的一份拷贝；但我们并不能保证 MEDLIB 的用户就一定能够得到 CONTAINER 库——况且这两个程序库是适用于两个完全不同应用领域的问题的。

获得、安装并且维护 CONTAINER 库（或者其他程序库）的一份拷贝都会花费用户的大量资金、时间和精力。如果这样的付出实在太大，那么潜在的用户将不会选择使用我们的程序库。即使获得、安装和维护我们程序库的开销比较低，用户有时也不太乐意使用我们的程序库。实际上，即使获得程序库的开销很小，获得其他可重用程序库这个必要性通常就会打

击许多潜在的、使用 MDELIB 程序库的用户的信心。

10.2.2　效率

重用其他程序库可能会导致我们程序库的效率低下。在这一小节里，我们将给出程序库设计者为了使重用其他程序库具有足够的效率而必须考虑并且分析的各种因素。

首先，考虑编译时间。如果我们重用 Set 类，那么#include Medical.h 的 MEDLIB 用户将会由于需要编译 Set.h 头文件，而使编译时间增加；甚至那些不需要使用 send_invoices 的用户也必须编译 Set.h 而浪费时间。因此，如果编译 Set.h 非常浪费时间，那么我们可以前置声明 Set 类来取代包含 Set.h 头文件（见 4.2.1 小节）：

```
Medical.h:
    template<class T> class Set;
    void send_invoices(const Set<Diagnostic>& diags);
    //...
```

现在的编译时间就应该是可以接受的了。

如果我们重用 Set 类，那么 Set<Diagnostic>就必须在用户程序中实例化，而实例化往往都会比较慢的（见 4.2.2 小节）。然而，如果我们在 MEDLIB 中预实例化（见 4.2.2 小节）Set<Diagnostic>，那么用户就不会负担实例化的时间开销。

现在考虑运行时间。虽然与为特定目的而设计的类相比，通用类 Set 的效率可能会稍微差些，但我们可以预期大部分用户一个月才会创建这些发票一两次；而且，创建这些发票通常都会涉及到读取操作、写入操作以及数据库存取，这些都是非常慢的操作；所以，使用 Set 的执行速度对整个运行时间几乎不会产生任何影响。因此，对整个运行时间而言，在 send_invoices 中使用 Set 的影响并不是比较明显。

类似地，因为大多数用户并不会经常创建发票，所以对用户而言，由于使用 Set<Diagnostic>而引起的内存使用量增加也应该是可以接受的。

因此，对于我们所分析的问题，可以得出这样的结论：重用 Set 将会是足够高效的。另一方面，如果试图对其他程序库的其他可重用代码作出和上面相似的分析，我们也可能得出重用导致效率不佳的结论。

10.2.3　冲突

重用其他程序库的另一个缺点就是可重用程序库可能会和我们程序库中用户使用的代码发生冲突（见第 6 章）。例如，假设 CONTAINER 库在全局作用域定义了一个 List 类，并且对于 MEDLIB 而言，它和全局作用域的 List 是毫无关联的，因为它从没有引用命名为 List 的对象。然而，如果我们的用户也在全局作用域定义了一个 List，那么 CONTAINER 库将会和用户代码发生冲突。在使用 MEDLIB 之前，必须修改这样的代码；而强迫用户修改他们的代码将会大大挫伤使用该程序库的积极性。

然而，一旦 C++编译器能够广泛实现名字空间特性（见 6.1.6 小节），那么程序库之间的

名字冲突将不再是一个棘手的问题。如果CONTAINER把List定义在一个特定名字空间里面，那么任何调用List的用户代码将不会和这个程序库发生冲突。

10.2.4 版本同步

程序库的内容会随着时间而不断更新，新的版本会增加新的功能，或者修改旧版本的已有功能，或者删除一些过时的功能。假设程序库X重用了程序库Y的部分代码。那么每个版本的X库的设计就需要和某些Y的特殊版本保持一致性，然而，并不是所有的Y版本都可以和X库协调一致的。因此，对于那些希望使用X的特定版本的程序员，就必须至少让一个Y库的版本可以和X的该版本协调工作。

为了简化我们的讨论，我们假设MEDLIB只有一个版本，并且这个版本已经经过测试可以用来和CONTAINER的Cm版本协调工作，如图10.1所示：

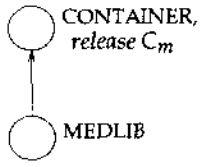


图10.1 MEBLIB依赖于CONTAINER的一个特定版本。

MEDLIB主要是设计用来和CONTAINER的Cm版本一起协调工作，并已经经过了协调性测试。

遗憾的是，MEDLIB的某些潜在用户可能没有CONTAINER的Cm版本，他们也不能得到这个版本；某些用户可能会在实现MEDLIB库之前，就已经拥有CONTAINER版本，因此所拥有的版本也就只能是早期的版本。对于拥有CONTAINER早期版本的用户，他们或者不愿意或者不能升级到Cm版本；因为他们代码的某些部分可能需要早期版本的某些特性，而在Cm版本中，这些特性并不存在；或者因为他们根本就不愿意支付升级费用。

另一方面，如果CONTAINER的新版本在MEDLIB对应版本发布之后才发布，就是说，我们的用户可能会拥有一个比Cm更新的CONTAINER版本。而拥有更新CONTAINER版本的用户可能并不希望降级到一个旧的版本。不妨把潜在用户拥有的CONTAINER版本称为版本Cu，如图10.2所示：

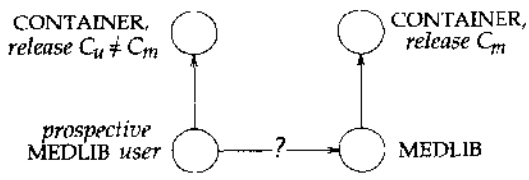


图10.2 版本同步问题。MEDLIB的预期用户可能会拥有一个不同于Cm的版本

那么对于具有不同于Cm的CONTAINER版本的用户，他们应该怎么做呢？好消息是

CONTAINER 的某些版本可以很好地和 MEDLIB 协调工作；坏消息是用户根本不知道他们拥有的那个版本是否属于可以协调工作的版本之一。

如果我们计划只分发 MEDLIB 的源代码，那么用户就只需要考虑 CONTAINER 版本之间的源代码兼容性（见 7.4 节）。如果 CONTAINER 的 Cu 版本和 Cm 版本是源代码兼容的，那么拥有 Cu 的用户也能和 MEDLIB 协调工作；这个结论是根据源代码兼容性的定义得出的。如果我们要分发 MEDLIB 的目标代码，那么用户就必须考虑 CONTAINER 版本之间的链接兼容性（见 7.5 节）。如果版本 Cu 和版本 Cm 是链接兼容的，那么 Cu 也可以和 MEDLIB 协调工作；这个结论是根据链接兼容性的定义得出的。

对于给定的程序库版本 Cu，程序员又是如何确定它和其他版本 Cm 是源代码兼容还是链接兼容的呢？这就涉及到下面 3 种情况（见图 10.3）：

1．如果 Cu 是 Cm 的早期版本，那么用户或许只能怪运气不好了。因为在编写 Cu 文档的时候，Cm 还没有存在，所以 Cu 的文档不会给出 Cu 是否会和 Cm 兼容与否的信息。不仅如此，实际上程序库的用户很少关心程序库早期版本是否会和晚期版本兼容，因此 Cm 的文档通常也没有说明这个兼容性问题。

2．如果 Cu 就是 C_{m+1}——Cm 的下一个版本，那么 Cu 的文档应该会给出它和 Cm 的兼容性信息。

3．如果 Cu 是 C_{m+1} 的后几个（多于 1 个）版本，那么 Cu 的文档可能（也可能不）给出 Cu 和 Cm 之间的兼容性情况。两者之间的版本数差得越多，Cu 的实现者给出这种兼容性信息的可能性就越小。如果 Cu 是 Cm 后续版本之后（即 2 个或 2 个以上）的版本，那么 Cu 的文档通常都不会说明上面的兼容性问题。（实现者通常都无视我们在 7.8 节的建议。）

那些不能确定 Cu 和 Cm 兼容与否的用户必须采取稳妥的做法。他们应该假设 Cu 和 Cm 是不兼容的，因此 Cu 就不能安全地用于 MEDLIB 库，把这个结论应用于图 10.3，我们将得到图 10.4。

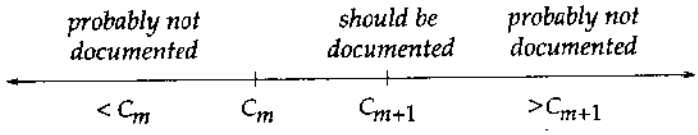


图 10.3　与 C_m 兼容的版本是否需要提供兼容性文档

图 10.4 看起来是非常令人失望的。它表明了，不但程序库的开发者必须知道他们开发的版本可以使用哪些其他程序库版本，而且用户还必须不辞劳苦地确认他们使用的程序库版本必须和所要求的版本保持同步。

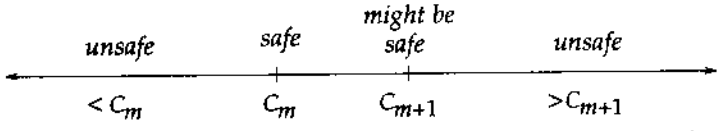


图 10.4　CONTAINER 的哪些版本可以被安全地用于 MEDLIB 库

乍看来，希望是那样的渺茫。然而，典型的程序库厂商中都会让几个程序库可以同时使用，但一般不超过 3 个。因此，在 MEDLIB 发布之前，它的开发者应该用所有可以使用的 CONTAINER 版本对 MEDLIB 进行测试。不仅如此，除非 MEDLIB 使用了 CONTAINER 的某些不清楚的特性（这是非常不明智的），否则 MEDLIB 应该可以使用 C_{m+1} 版本——因为程序库通常只会增加功能，而很少减少功能。

某些 C++实现通过提供程序库的版本管理机制（见 7.6 节，即在创建的时候给出一个版本号），可以帮助用户处理版本同步问题。程序库提供者通过使用这种版本管理机制，来指定给定的版本和程序库其他的哪些版本是兼容的。

10.3 自含式（Self-Contained）程序库

对于重用其他程序库而带来的问题，一种避免的方法是让程序库是自含式的。自含式程序库定义和实现它所需要的所有功能，而没有使用其他程序库，除非其他程序库的接口非常地好——例如标准的 C 和 C++程序库（见 9.8 节）。

我们可以用两种方法来开发自含式 MEDLIB。可以购买在 MEDLIB 中使用 CONTAINER 的权利，和把 CONTAINER 作为 MEDLIB 分发产品一部分的权利。或者，我们可以编写自己的集合类。这里我们只考虑第二种方法。假设我们把自己编写的集合类命名为 Medset，这是为了避免和 CONTAINER 的类 Set 发生混淆：

```
template<class T>
class Medset {
    //...
};
void send_invoices(const Medset<Diagnostic>& diags);
//...
```

一旦我们把 MEDLIB 实现为自含式的，用户就无需再忧虑需要找一个和 MEDLIB 能够协调工作的 CONTAINER 版本（或任何其他的程序库）。不仅如此，用户也不会遇到名字冲突问题，这里的名字指的是在其他程序库而不是 MEDLIB 库使用的名字。当然，把程序库实现为自含式程序库也存在某些缺点，我们将在接下来的 10.3.1 小节到 10.3.4 小节讨论这个问题。

10.3.1 实现困难

与重用其他程序库相比，自含式程序库的实现通常都比较困难。如果选择了设计自含式程序库，那么就必须自己提供所有的功能。（前面已经指出，实现的容易性是我们选择重用已有程序库的主要动机。）

通过提取（scavenging，见 1.1.1 小节）——这里指把某些已有的集合类（如 CONTAINER 中的 Set 类）代码拷贝到我们的程序库中来，我们可以简化实现自含式程序库的工作。当然，只有和代码所有者签有专利使用权转让协定，我们才可以把那些代码拷贝到我们的程序库中。

然而，如在 1.1.1 小节所述那样，提取代码是很困难的，所以不应该轻举妄动。

10.3.2　使用困难

与重用其他程序库相比，自含式程序库的使用将比较不方便。对于那些既使用自含式程序库 MEDLIB，又使用 CONTAINER 程序库的用户，他们就必须理解两个集合类（Set 和 Medset），而不使用自含式程序库的用户只需要理解一个集合类（Set）就足够了。

自含式程序库的深层次的复杂因素涉及到具有转型功能的函数。考虑 MEDLIB 的一个自含式版本，它定义了自己的 Medset 集合类，假设 MEDLIB 的一个用户还使用了 CONTAINER 库。那么如果这个用户尝试传递一个包含 Diagnostics 的 Set 对象给我们的 send_invoices 函数：

```
Set<Diagnostic> diags;
//...
send_invoices(diags);   //error
```

对于用户而言，这个调用看起来是无可挑剔的——diags 代表诊断测试的集合，send_invoices 用来为给定的诊断测试集合创建发票。遗憾的是，对 send_invoices 的这个调用却会导致编译失败，因为 send_invoices 需要接收一个 Medset<Diagnostic>类型的参数，而不是 Set<Diagnostic>类型的参数，而且并没有定义任何从 Set<T>到 Medset<T>的转型。

有两个位置可以用来定义从 Set 到 Medset 的转型。首先，Medset 可以提供一个单参数的构造函数来实现转型：

```
template<class T>
class Medset {
public:
    Medset(const Set<T>&);
    //...
};
```

另一种方法是，Set 可以提供一个转型函数来实现转型：

```
template<class T>
class Set {
public:
    operator Medset<T>() const;
    //...
};
```

然而，Set 中的这个转型函数存在的可能性不大，因为 CONTAINER 是一个通用的容器类程序库，它的设计者通常都不知道 MEDLIB 的存在，况且 MEDLIB 还只是一个应用于特定领域的程序库。即使他们确实知道存在 MEDLIB，甚至希望提供这个转型函数，但在不破坏 Set 类自含式性质的条件下，他们还是不能这样做（即提供这个函数）；况且对一个通用的容器类而言，自含式性质是非常重要的。

因此，假设我们要在 MEDLIB 中提供第一个转型。为了保持 MEDLIB 库的一致性和完整性，我们是否还需要提供所有其他已知容器类程序库中集合类到 Medset 的相似的转型（即类似于 Set 到 Medset 的转型）呢？如果我们都提供了这些转型，那么 Medset 将会有太多太

多的构造函数。那么我们是否就应该只提供这一个转型呢？

这一次，我们将不需要做出上面这个进退两难的决定；因为即使我们确实希望提供上面给出的在 Medset 中的转型，但是实际情况并不允许我们提供这个转型。考虑下面这个转型的实现：

```
#include<Set.h> //Set.h 来自 CONTAINER 程序库。
template<class T>
Medset<T>::Medset(const Set<T>& s) {
    //...
    //存取 s 中的某些成员。
    //...
}
```

为了从一个 Set 对象构造出一个 Medset 对象，我们必须存取 Set 的某些成员；为了存取 Set 的成员，我们必须#include CONTAINER 中包含 Set 定义的头文件（Set.h），但这将导致 MEDLIB 不再是自含式程序库。

因为上面两种具有转型功能的函数都是存在的，所以对那些既使用 MEDLIB 又使用 CONTAINER 的用户，如果希望使用包含 Diagnostics 的 Set 对象来作为 send_invoices 的调用参数，那么就必须自己实现一个从 Set 到 Medset 的转型函数：

usercode.c:

```
template<class T>
Medset<T> set2medset(const Set<T>& s) {
    //...
}
```

使用这个显式转型函数，用户就可以如下编写代码：

```
Set<Diagnostic> diags;
//...
send_invoices(set2medset(diags));
```

毫无疑问，对用户而言，实现一个从 Set 到 Medset 的转型函数还是比较方便的。

10.3.3 效率

自含式程序库可能太低效了。例如，调用转型函数 set2medset 将会耗费运行时间；而那些使用非自含式版本的 MEDLIB 代码既不需要实现转型函数，也不需要执行这个转型函数。自含式程序库还会增加用户的代码大小；如果 MEDLIB 定义了自己的集合类，而用户程序使用了另外的集合类，那么用户的可执行文件将可能由于包含两个集合类而增大不少。

如果每个程序库都是自含的，那么使用几个程序库的大型程序的代码大小将会非常庞大。随着对程序库依赖性的不断上升，对于应用程序和程序库本身而言，代码庞大问题将会不断恶化。

10.3.4 隔离

在把自己从对其他程序库的依赖性中隔离开来的过程中，自含式程序库也使自己没办法

实现某些对程序库有益的修改。例如，假设 CONTAINER 的提供者发布了一个新的版本，版本中包含了效率更快的 Set 类；这时，如果 MEDLIB 是自含式程序库，那么用户就不能从这个有益改变中受益。

然而，通过设计者在自身程序库中对类进行控制，这个不足是可以弥补的。例如，如果 MEDLIB 是自含的，我们可以在 CONTAINER 的开发者发布更快 Set 版本之前，就开发出自己更快的 Medset 版本。实际上，如果 MEDLIB 是自含的，我们就可以根据 MEDLIB 的特定用途，自定义地优化 Medset。而对于某些其他程序库的特殊用途，CONTAINER 的设计者很少对 Set 进行优化。

10.4　总结

使用其他的程序库可以使实现更加容易，但也带来了这样一些问题：获得程序库的问题、冲突、版本同步和可能的效率问题。

自含式程序库可以避免上面这些问题，但它通常会使程序库的实现容易性、使用容易性和效率受到损害。某些程序员也使用以别的方式重用过的程序库。自含式程序库可能会要求程序员懂得多重接口，并且编写显式调用的转型函数，从而影响了使用的容易性。不仅如此，自含式程序库还可能会导致用户的可执行代码过度膨胀。最后，当自含式程序库和其他程序库隔离的时候，将会出现期望的和非期望的两种效果。

10.5　练习

10.1　假设你在开发练习 1.2 和 7.1 所讨论的 Path 类，请考虑下面的成员函数，它使用一个 List 类来描述 Path 的目录：

```
class Path {
public:
    List<Path> expand_wildcards() const;
    //...
};
```

函数 expand_wildcards 将会搜索程序所在机器的文件系统，并返回符合（指根据某种已经定义的规则集合）给定 Path 的文件列表。

相对于使用你自己实现并为 Path 列表而优化过的专用链表类，上面的程序重用了一个具有商业通用的 List 类，就效率方面，请对这两种情况给出你的分析（根据 10.2.2 小节的分析方法）。

10.2　假设我们程序库重用 CONTAINER 程序库的唯一事物就是 Set 类，而且我们重用 Set 类的唯一方式就是作为 send_invoices 函数的参数类型（见 10.1 节）。即使我们已经在自己的程序库档案文件中预实例化了 Set<Diagnostic>，用户仍然需要获得 CONTAINER 程序库，

请解释原因。

10.3 假设我们使用 Set<Diagnostic>的唯一地方是在程序库的实现文件中（就是说，并不是在公共头文件中）。

a. 预实例化 Set<Diagnostic>是否可以使我们程序库的用户不需要获得 CONTAINER 程序库？

b. 预实例化 Set<Diagnostic>是否可以避免 10.2.3 小节所说的冲突？请给出你的理由。

第11章

文档编制

这封信我写得比通常长了许多，因为我不具备使它变精炼的时间。

—Blaise Pascal

对于程序库的成功，好的文档是至关重要的。在这一章里，我们将讨论哪些文档应该附加到程序库里——设计文档、使用指南和参考手册。我们最后总结出建议：为参考手册需要记录的东西编写使用指南和说明书。

11.1 文档编制和重用性

没有经过适当文档化的代码不能成为可重用代码。一个设计得很好的程序库，如果文档编制得好，那么将会是一个文档齐全的程序库，否则会功亏一篑。对于希望快速学会如何使用程序库的用户而言，好的文档也是必不可少的。就算是小规模的程序库，也不应该忽视文档的编写；即使程序库只是提供很少的、很容易理解的特性，也是如此。因为对于任何一个程序库，如果设计和实现相同功能新程序库的时间比学习这个程序库的时间还要少，或者相等，那么这个程序库将会无人问津。

文档应该是清楚而且完整的。即使对你而言，程序库的设计和某些要被使用的特性都一目了然，但对于你的用户情况往往就不是这样；因为如果你已经花费了几个月甚至几年的时间设计和实现这个程序库，那么你的看法和初次接触这个程序库的用户的看法肯定是不可同日而语的。

编制高质量的文档通常都会占用设计、实现和测试程序库很大的一部分时间。之所以需要编制好的文档，主要是由于开发可重用代码的开销要比开发单一用途代码的开销多很多。

当设计和实现程序库的时候，程序库设计者就应该编制文档。编制文档的过程往往可以

揭示提高代码质量的方法。即使为了简化程序库的设计，而延迟编制文档也是不正确的。Donald Knuth[Knu89]观察到设计和文档的编写之间具有密切的联系：

新系统的设计者不仅仅应该是系统的实现者和第一位用户，还应该编写第一份用户手册……如果我没有完全参与到这些工作来，那么简直就不会有这么多改善；因为我从不会对所作的设计进行充分的思考，也不可能知道这些设计为什么重要。

每一个可重用的C++程序库都至少应该附带设计文件、使用指南和参考手册。我们将在11.2节到11.4节讨论这些内容。

11.2 设计文档

在这整本书中，我们看到可重用代码的设计者总会碰到一些几乎令人茫然的设计选择。大多数设计决策都会使生成的程序库对某些潜在用户更有用处，而对其他的用户用处减少。因此，在设计文档里，程序库设计者应该解释所作出的决定、为什么做出这些决定、程序库主要面向的用户和程序库给这些用户提供了哪些功能。特别地，C++程序库的设计文档至少应该回答下列问题：

- 程序库提供了哪些类？
- 继承体系如何？
- 程序库的所有类是否都是nice类（见2.3节）吗？
- 程序库的效率如何？
- 在什么条件下，程序库才是可扩展的？
- 该程序库是否使用了其他程序库？如果是，那么该程序库已经测试了其他程序库的哪些版本？
- 程序库是否具有向前或者向后的源代码兼容性、链接兼容性、运行兼容性或者进程兼容性？
- 程序库的错误处理机制如何？
- 程序库的可移植性如何？
- 该程序库是否会和其他程序库发生冲突？

另外，设计文档还应该特别地指出该程序库所适用的问题领域。

11.3 使用指南

使用指南给出应该如何使用一个程序库。使用指南的用途是在于帮助用户可以学会使用这个程序库，因此它省略了许多关于程序库的高级或者特殊用法方面的细节。大多数程序库应该提供一个用于整个程序库的使用指南，而且还应该为程序库的每个类或者相关类的集合提供单独的使用指南。

编写使用指南几乎就是一种艺术。学习这种艺术最好的方法就是阅读他人写得好的使用指南，然后仿照写出自己程序库的使用指南。在这里，我们不是列出某个使用指南的内容，而是在下面的几节里，提供使用指南编写风格的一些建议。我们的使用指南例子来自于下面类所需要的使用指南：

```
class Regex {
public:
    Regex(const char* s);
    //...
    bool match(const char* s);
    bool match(const char* s, String& t);
};
```

Regex 是一个正则表达式。上面所给的构造函数使用给定的 s 指向的以 null 结尾的字符串，创建了一个 Regex，用来表述一个正则表达式。例如，下面的正则表达式匹配任何含有零个或多个数字的字符串：

```
Regex r("[0-9]*");
```

11.3.1　对读者的背景知识了如指掌

在编写一份使用指南之前，应该了解读者的知识状况如何。考虑下面的关于 Regex 类的部分描述：

下面的语句声明了一个 Regex 实例，并初始化为一个空的正则表达式：

```
Regex r;    //默认值是: ""。
```

对于 C++新手，这个描述就是恰到好处的。然而如果面对一个已经对 C++有所了解的读者，就不需要告诉他们："声明了一个 Regex 实例"。而高级读者则可能会更喜欢简洁的描述：

Regex 对象的缺省值是一个空的正则表达式：

```
Regex r;    //""
```

一旦你选择了某种层次的描述，在程序库的整个使用指南中，就应该始终如一地使用这种层次来描述。

你还需要考虑用户 C++之外的其他知识；例如，下面的描述就是假设读者知道 egrep 和正则表达式的具体内容：

Regex 是 egrep-style（egrep 样式）的正则表达式。

某些用户可能不知道 egrep 是在某些系统中模式匹配（pattern-matching）程序的名称。对于那些缺乏正式计算机或者数学知识培训的读者而言，正则表达式的概念也是陌生的。

如果用户缺乏理解你的文档或者使用你的程序库所需要的相关背景知识，那么你就需要提供所需的知识，或者提供有关这些信息的参考文献。

11.3.2　用抽象的观点来编写

编写的函数行为应该依据它们对对象抽象值的影响，而不是依据它们的底层实现。下面

的表述就展示了实现细节：

函数 match

```
if(r.match("Earl Grey"))    //...
```

将会高效地匹配给定的字符串和 Regex 内部编译后的形式。

相反，下面的表述只是基于涉及到的对象的抽象值做出解释，就是我们所提倡的：

函数 match

```
if(r.match("Earl Grey"))    //...
```

在给定字符串和正则表达式完全匹配时，就返回真值。

偶尔，某些使用指南的读者会对底层实现的讨论充满兴趣。那么如果确实需要讨论的话，意思应该非常地明确：

Regex 的实现是通过把给定的正则表达式编译成一个内部的形式，该形式可以提高匹配的效率。

编写者也可以考虑把所有对实现的引用（解释）做成一个单独的“实现注释”文件，并把它放置在使用指南的最后部分。

11.3.3 先解释普通用法

在一个使用指南中，并非所有的功能都必须加以解释，更不用不分轻重地解释。指南通常只需要给出让用户知道如何开始操作的知识，和用户经常希望使用的功能等方面的知识。

某些函数可能会有两方面的功能：经常使用的功能和偶尔使用的功能。对于这些函数，在使用指南中，不能一开始就讨论偶尔使用的功能。例如，对于 match 类的某些高级使用，我们就应该留到后面来描述，如下所示：

前面，我们给出了函数 match 可以用来判断 Regex 是否和某个字符串匹配。另外，当增加一个 String 参数来调用这个函数时，它将返回参数中匹配的子串，代码如下：

```
String matching_substring;
if(r.match("Earl Grey", matching_substring))    //...
```

如果程序库中两种用法的几率是相等的，那就以它们在典型用户代码中出现的顺序来对它们进行解释。例如，在进行匹配之前，用户需要构造一个 Regex 对象，那么我们就应该先说明构造：

首先，使用想要的模式构造一个 Regex：

```
Regex r("[1-9]*");
```

接下来，对照目标字符串匹配这个 Regex 对象：

```
if(r.match("Earl Grey"))    //...
```

11.3.4 一次只解释一个事物

一次解释多种想法的愿望是好的。但是，先考虑下面不明智的表述：

通过调用 match 函数，可以使用目标字符串来匹配 Regex 对象：

```
String matching_substring;
//可选的第 2 个参数
if(r.match("Earl Grey", matching_substring))    //...
```

如果匹配成功，那么这个调用将返回真值，String 参数 matching_substring 也将被设置为在目标字符串（和 Regex）所匹配的子串。

如果这个表述是出现在指南中，那么一大部分读者将会对后面这个句子产生疑惑。

11.3.5　解释用法，不解释设计思路

可重用代码的指南应该是只讲授如何使用代码，而不解释为何如此设计代码。所以，我们不应该这样编写指南：

用户经常需要知道匹配子串的索引和长度，为了能够最小化函数 match 的重载实例的数量，我们可以提供下面的接口：

而应该这样陈述：

那些需要匹配子串的索引和长度的用户，可以调用下面的函数：

对于任何关于程序库设计的内容，只有在这些内容对用户理解和记住如何使用这个程序库有帮助时，我们才应该在指南中声明它们。例如，如果用户非常熟悉 UNIX 系统，那么下面所包含的关于 Regex 设计的内容就很有帮助：

只要在可以使用的情况下，Regex 提供的功能和 UNIX 的 egrep 命令的行为才是一样的。

11.3.6　简单清楚地编写

风格的独创性在文学作品中是很受赞赏的，但就技术文档的编写来说，坚持使用统一的格式是更可取的。使用指南的目的在于向用户展示如何使用程序库，任何妨碍这个目的的编写风格都是应该避免的；特别地，

- 避免过于巧妙。　即使很少的巧妙之处都有可能令用户感到疑惑，这并不是因为用户缺乏理解这些巧妙之处的智商，而是因为当用户阅读使用指南的时候，通常都还要忙于其他的事情——例如，学习如何使用程序库。
- 避免使用幽默。　除了很少一小部分以外，幽默都不应该用于使用指南。要想通过幽默来找到希望学习的内容，那常常会令人无功而返。
- 保持简洁。　对于一个使用指南，如果读起来好像是意大利的伟大作家但丁的著作，那肯定是了不起的文学巨著，但肯定不是一份写得好的使用指南。

11.3.7　准确地使用语言

程序库的文档编写者应该准确地运用他们日常使用的语言（如英语、日语、俄语等等）。下面有一个反例正好可以解释这一点。当处理如下一个语句的时候：

```
x = (f(y) ? w++ : throw 19, 1);
```

C++编译器的 alpha 版本可能会给出下面的警告:

```
warning: throw is not last oprand of comma expression, entire
expression will not be evaluated.
```

这个编译器实现者想表达的意思是，并不是所有的表达式都会被求解；然而，上面的警告表达的意思是没有表达式会被求解。对于那些熟悉 C++的用户，可能会觉得这只是一个语言表达错误；但对那些对 C++不太精通的用户，却很容易会造成误解。从这个例子可以看出，让优秀的编写者来审阅你的文档是非常重要的。毫无疑问，上面的信息对编写它的人而言，几乎是完美的，因为要查找出自己编写的文档的错误往往是很困难的。

11.3.8 使用普遍接受的术语

如果有普遍接受的术语可以使用，那么就应该使用这些术语，而不要使用你自己杜撰的术语。例如，应该坚持下面已经确定的描述

T::T(const T& t)是类型 T 的拷贝构造函数。

而不是

T::T(const T& t)是类型 T 的克隆函数。

即使你认为你的术语更好，也应该使用已经确定的术语，这样才算是为你的读者考虑。如果没有确定的术语可以使用，而你又必须杜撰一个术语；那么在第一次使用的时候，就应该给出准确的定义。例如：

在下面，我们将称 T::T(int i)为类型 T 的 int 构造函数。

当我们写的内容是跟 C++有关的时候，就应该使用 C++的术语；特别地，不能使用其他面向对象语言的术语来代替 C++术语或者与 C++术语混合使用。其他语言的术语没有什么问题，主要是因为 C++程序库使用指南的读者，他肯定不希望花时间来细细分辨两个属于不同语言的词汇。譬如，某些语言就把 C++中的成员函数称为方法，把 C++中的调用成员函数称为发送一个消息。

11.3.9 深刻理解重载的术语

对那些具有多个含义的单词，应该格外小心。考虑下面 instance 的 5 个用法：

1. 一个类 T 的 instance；
2. 类模板 T 的 instance T<int>；
3. 函数模板 f 的 f(int) intstance；
4. 重载函数 g 的 g(int) instance；
5. 普遍规则的 instance。

要想深刻理解所有这些 instance 的用法是很困难的。其中前面 3 个用法可以使用下面更好的语言来表达：

1. 一个类 T 的对象；

2. 类模板 T 的特化 T<int>；

3. 函数模板 f 的 f(int)特化。

（前言中有关于特化这个术语的讨论。）

11.3.10　给出合法的、无错误的代码

使用指南中的代码不应该存在无意的错误。如果出现语法或者逻辑错误，那只能说明没有对代码例子进行严格的测试，代码编写者是非常粗心和懒惰的；从而也会导致用户对程序库或使用指南失去信心，或者对两者都失去信心。

例子中的代码也不应该包含有意的错误。通常，向用户展示应该如何做，要比展示不应该怎么做有效得多。因此，我们应该这样说：

```
f(1);    //用一个非零参数调用 f。
```

而不是：

```
f(0);    //传递一个 0 值给 f 是错误的。
```

然而，有时给用户展示某些错误，可以帮助用户理解和避免这些错误。例如，

如果用户指定了一个无效的正则表达式，那么将会抛出一个异常：

```
Regex r("r[0-9");    //错误的正则表达式
                     //抛出 Regex::Invalid 异常。
```

在例子中务必要标注错误的代码。例如，如果上面的例子出现在用户指南中，而没有后面的注释行，那么那些快速浏览指南的用户可能发现不到这段代码是错误的，从而也就会在他们自己的代码中犯下相同的错误。

11.3.11　保持简短的代码段

在指南中应该尽量保持简短的代码段（通常是 20 行以内，但肯定要在一页以内）。代码例子越长，用户跳过这段代码或者产生疑惑的可能性就越大。如果你要说明的观点需要很多行代码，那么应该用文替的文本把这些代码分开。例如，我们不应该这样写：

```
用户必须执行下面的操作：
        //很长的一段代码
        //...
```

而应该这样编写：

```
用户首先应该做下面的事情：
        //短代码段
接下来必须这样做：
        //又一个短代码段
```

11.3.12　避免使用太大的函数

无论是指南中的代码例子，还是现实的代码，使用太大的函数都是很不好的风格。指南编写者不应该把几个很长的代码块都组合在一个大的函数里面。考虑下面的代码：

用户必须如下提供函数 f:

```
void f() {
    while(some_condition)
        //...
}
```

while 循环的代码实体如下:

```
if(another_condition) {
    //...
}
else {
    //...
}
```

当另一个条件为真时，f 还会调用下面代码:

```
switch(expression) {    //...
}
```

如果这种解释的方式占用了一页或者好几页，那么 f 就可以被看作一个太大的函数；即使在指南中不使用这个函数，也是如此。用来描述函数 f 的正确的方法应该是把它分散成几个小函数:

用户必须如下提供函数 f:

```
void f() {
    while(some_condition)
        g();
}
```

函数 g 的代码如下:

```
void g() {
    if(another_condition)
        h(expression);
    else
        //...
}
```

函数 h 的代码如下:

```
void h(int expr) {
    switch(expr)
        //...
}
```

这种模块化的函数，对于代码的读者和文档的读者都是更加容易理解和接受的。

11.3.13 提供在线实例

在网上应该提供指南的大部分代码，并作为程序库发布工作的一部分。当用户知道这些实例代码可以通过与程序库编写者的机器联机获得时，就可以下载这些代码。因为如果需要测试代码，用户通常是不愿意自己输入代码的。

11.4　参考手册

程序库的参考手册应该以完整、准确、详细的方式来文档化整个程序库。大多数 C++程序库主要是由类的集合组成的；因此，程序库文档的编写者就应该知道如何完整、准确地文档化一个类。总而言之，类 X 的文档应该包含下列几个方面：

1. 告诉用户需要#include 哪些文件才能得到类 X 的定义。除非有充分的理由采用其他方式，否则程序库的每个公共头文件都应该#include 或者声明（见 4.2.1 小节）它所需要的对象。采取这种方法后，我们就不会为了得到某个类的定义，而需要#include 多个头文件。
2. 定义类 X 的抽象。
3. 给出类 X 的语法接口。
4. 描述 X 接口中每个函数的语义。
5. 声明任何对模板参数的限制条件。

其中，第 1 点应该是不言而喻的；我们将在后面几个小节里详细阐述后面 4 点。

11.4.1　抽象化

清晰定义每个类的抽象，不管多么过分强调，都是合情合理的。为了可以巧妙地使用某个类，用户必须先知道这个类所模拟的抽象。遗憾的是，现今存在的文档，大部分都未能正确地文档化类的抽象。例如，现存的文档大体都是讲述下面的内容，而根本就不是定义类 Regex 的抽象：

- 对于操作正则表达式，类 Regex 是非常有用的。
- Regex 应该用于执行正则表达式的匹配。
- Regex 为程序员提供了可以更容易地处理正则表达式的方式。

对于上面的语句，虽然每一个都是真的，但是没有一个定义了 Regex 的抽象。

究竟要用何种层次的准确性来定义类的抽象要取决于这个类本身和类用户的背景。例如，如果这个类是 Regex，类的用户都是非常机敏的，那么只要说“Regex 对象是正则表达式”就已经足够了。

11.4.2　语法接口

在参考手册中，应该给出类 X 的定义的哪一部分呢？当然不是整个定义了，因为典型类的定义大部分都是关于实现细节的内容，用户根本就不需要关心这些内容。考虑下面的代码：

```
class X {
    friend ostream& operator<<(ostream& o, const X&);
    //...
};
```

这个输出操作符是类 X 的友元，而这对 X 的用户根本就不产生影响；因此，在参考手册

中，我们就不需要给出这个友元声明（或者任何友元声明）。（当然，我们在某些地方应该说明这个操作符是一个输出操作符）。

显然，对于任何一个类，我们应该给出所有的公共基类与保护基类和类本身的公共成员和保护成员：

```
class X : public Document_this_base {
public:
    void document_this_function();
protect:
    void and this_one_too();
    //...
};
```

对于一个函数成员（或者说成员函数），它的语法接口就是其原型，包括返回类型、参数类型和函数是否是 virtual、const 或者 static 的。

1．自动产生的成员函数

如前所述，C++的定义说明，如果类 X 没有声明任何构造函数，那么编译器将会产生默认的公共构造函数。类似地，如果没有声明拷贝构造函数、赋值运算符和析构函数，那么编译器也会产生这些函数。

程序库的参考手册可以用两种方式来处理这些自动产生的函数：

- 可以不提及这些函数，而让用户自己推导出它们的存在。

例如，如果类 X 的文档只包含了下面的公共接口：

```
class X {
public:
    X();
    X(const X&);
    void f();
};
```

那么用户可以推导出，编译器为类 X 自动产生了一个赋值运算符和一个析构函数。

- 另一种方式是，参考手册应该显式地给出这些自动产生的函数，即使这些函数并没有出现在类的声明代码中：

```
class X {
public:
    X();
    X(const X&);
    const X& operator=(const X&);
    void f();
    ~X();
};
```

实际上，赋值运算符和析构函数是由编译器自动产生的这个事实只是一个实现细节，对用户通常都不会有任何影响。

这两种方式中的每一种都是可以使用的。参考手册应该指出究竟是如何文档化这些自动

产生的函数的。另外，如果选择了某一种方式，那么整个文档应该自始至终都使用这种方式。

2. 私有成员

文档编写者最初会冲动地认为，私有成员并不需要在参考手册中给出。然而，下面情况下的私有成员却需要给出：

- 如果不给出私有声明，就会自动产生一个私有成员函数。
- operator&的私有声明。

例如，我们应该文档化下面类的私有成员以表明拷贝和获取 X 对象的地址都是被禁止的：

```
class X{
private:
    X(const X& x);
    X* operator&();
public:
    X();
    //...
};
```

那么其他的私有成员呢？例如，如果 X 声明了一个私有成员函数 f，那么参考手册是否应该给出 f 的内容呢？如 6.1.5 小节所指出的，C++在存取保护检查之前的二义性检查会使程序库的私有名称在用户代码中导致二义性。而且，文档化所有的私有名称也会使用户的文档显得混乱。因此，大多数程序库都没有文档化私有名称，但前面所提及的特殊情况除外。由于类似的原因，大多数程序库的参考手册也没有给出私有基类。

11.4.3　函数语义

参考手册应该在需要文档化的类的语法接口中，声明每个函数的语义（除了正规函数，我们将在后面讨论这一点）。于是，编写者应该遵守下面的约定：

- 如在 11.3.2 小节讨论的那样，语义的解释应该依据所给对象的抽象值，而不是对象的底层实现。
- 当文档化虚函数的时候，还应该指定它的继承语义（见 3.3 节）。
- 当返回类型为指针或引用的函数被文档化时，应该指出这个返回值的生存期（见 4.4.3 小节）。
- 如果函数抛出异常（见 5.3.4 节），还应该指定每个异常的类型和这个异常的抛出条件。

而正规函数（见 2.2 节）的语义是不需要文档化的。例如，根本就没有必要这样文档化 X 的析构函数：它释放 X 在内部分配的内存空间。因为如果 X 的析构函数并没有释放 X 分配的内存空间，那么 X 的实现就是不正确的，我们还必须回过头来修正这个错误。

如果一个类的设计是正确的，但文档编写者觉得还需要文档化类的某一个正规函数，那么可能是因为当初没有定义好这个类的抽象。例如，假设程序库提供了下面的类和相应的函数：

```
class File {
```

```
public:
    File(const char* path);
    //...
};
bool operator==(const File& file1, const File& file2);
```

假设编写的文档如下：

File 描述文件的名称，构造函数创建了给定的函数名称。如果文件 f 和文件 g 的名称相同，或者是同一个文件的引用，那么相等运算符将返回真值。

如上所述，相等运算符的语义并不是正规的；我们可以通过纠正抽象的定义，来使这个语义变成正规语义：

File 描述底层操作系统中的一个文件。它的构造函数创建了一个 File 对象，用来描述给定的 path 引用的文件。不同的路径可以引用相同的文件。

如果这个相等运算符具有正规语义，那么当 f 和 g 具有相同的抽象值时——就是指 f 和 g 引用相同的文件，那么相等运算符将返回真值。如果有了这样一个编写得好的 File 抽象的定义，那么相等运算符的行为就不言自明了。

11.4.4 模板参数约束

任何对类模板参数或者函数模板参数的约束，都应该在参考手册中进行文档化。假设程序库提供了一个 sort(排序)模板：

```
template<class T>
void qsort(T* t, int n);
```

这个函数从 t 开始的元素为 n 个 T 类型值的数组进行排序。假设 qsort 函数使用 < 运算符来比较数组元素：

```
template<class T>
void qsort(T* t, int n) {
    T* t1;
    T* t2;
    //...
    if(*t1 < *t2)
        //...
    //...
}
```

当调用 qsort 的代码编译时，对于类型 T 而言，qsort 中 < 运算符的使用必须是合法的。那么文档就应该给出这个约束，也许就是这样的：

T 必须是这样的一个类型：如果 x 和 y 都是类型 T 的对象，那么表达式 x < y 是合法的，并且在 x 被认为小于 y 时，返回真值。

正确地文档化模板参数的约束是困难的。譬如，qsort 的文档编写者可能就想如下编写这个约束：

类型 T 必须提供一个 < 操作。

然而，这个语句给出的约束远大于代码所需要的约束。考虑下面的代码：

```
class X {
public:
    operator int() const;
    //...
};
```

即使类 X 并没有提供 < 操作，但对 X 类型的数组调用 qsort 函数将是合法的，因为表达式*t1 < *t2 会先把*t1 和*t2 转化为 int 类型的对象，然后再应用内建类型 int 的 operator <。

11.5 总结

好的文档对可重用代码是至关重要的。C++程序库附带的文档至少应该包括设计文档、使用指南和参考手册。

程序库的设计文档应该讨论程序库设计环节的每个重要决定，为什么要做出这样的决定，为谁（哪些用户）而做出这样的决定和程序库为用户提供了哪些功能。

使用指南的编写应该是清楚和简单的，并且在编写时还要考虑程序库用户的背景。使用指南应该依据抽象值来讨论程序库的功能，而不应该基于程序库的实现来讨论这些功能。而且，指南中的例子包含的应该是合法、正确的代码。

程序库的参考手册应该定义程序库中每个类的抽象，展示每个类的语法接口，给出每个类的接口函数的语义，并且提供模板参数的约束。

11.6 练习

11.1　考虑 Path 类的下面版本，实际上，这个类在练习 1.2、练习 7.1 和练习 12.14 都有讨论：

```
class Path {
public:
    enum Style { UNIX, Windows, VMS };
    Path();
    Path(const Path& p);
    Path(const String& str, Style style = UNIX);
    const Path& operator=(const Path& p);
private:
    void canonicalize();
    Style the_style;
};
ostream& operator<<(ostream& o, const Path& p);
```

请问，在参考手册中，上面的哪些函数应该进行文档化？

11.2　(*) 在 11.4.2 小节，我们讨论了在程序库的参考手册中，私有成员函数是否应

该文档化的问题。

a. 给出一个合理的用户代码例子，这段代码因为程序库中存在一个私有基类，而导致不能运行。

b.（*）在什么条件下，程序库才需要文档化它的私有基类？为什么？

11.7 参考文献和相关资料

很好地运用语言（或英语、日语及其他语言）要比大多人想象的困难许多。编写好的文档就更加困难了。Strunk 与 White 的作品[SW79]、Dupre 的作品[Dup95]和 Hodges 与 Whitter 的作品[HW67]都给出了许多关于如何运用英语的很有价值的建议。

第12章 其他话题

人类任何事业的起点和终点都是凌乱不堪的，小到建筑一栋房屋，编写一本小说，摧毁一座小桥，大到结束一次航海旅行，都是如此。

——*John Galsworthy*

在这一章里，我们将讨论其他几个重要的程序库设计话题。首先，我们阐述了静态初始化问题，并且给出了遇到这种问题时实现程序库的几种技术。在定义了局部化开销原则之后，我们还给出了警告，如果没有细心谨慎地设计程序库，那么就有可能会违反这条原则。接下来，在可重用类里面，容器的使用是非常普遍的。因此，我们提出了两个和容器相关的话题：区分内生容器与外生容器和设计迭代器。然后，我们讨论了类耦合的优缺点——类耦合是指在同一个程序库里，在一个类的内部使用其他的类。最后，我们给出了C++程序库设计者应该如何把某些决定留给用户，从而避免作出左右为难的设计决定。

12.1 静态初始化问题

C++继承了C语言静态对象初始化的概念（静态对象指存储在静态存储区的对象，可以用static关键字声明，也可以不用这个关键字声明）。例如，下面的定义语句如果出现在全局作用域，那么这条语句就会把值0赋给静态对象i：

```
int i = 0;
```

C语言限制了只能把静态对象初始化为常量表达式（指在编译期或链接期可以求值的表达式），而C++在这一点上和C有区别，C++允许任意表达式都可以在初始化时出现。C++的这个特性也导致了静态初始化问题。在下面各小节里，我们将详细讲述语言定义应该如何来表示静态对象的初始化顺序，为何C++程序库的设计者会遇到这个问题，以及程序库设计

者应该如何处理静态初始化。

12.1.1 构造和析构的时刻

C++语言已经确定了程序中对象的构造和析构时刻。考虑一个含有构造函数和析构函数的类——例如，2.4.1 小节的专用的内存分配器类 Pool。当控制流程经过类型为 Pool 的堆栈对象 p 的定义时，p 就会被构造（创建）；当控制流程离开 p 的包含模块（指两个花括号之间的部分）那一刻，p 对象就会被析构（删除），这一点是确定无疑的：

```
{
    //...
    Pool p(20); //p 现在被构造。
    //...
}                          //p 现在被析构。
```

而且，在给定模块中的所有对象，它们都是以和构造时相反的顺序进行析构的。而对于在空闲存储区分配的对象，它们是在分配空间的时候被构造的，并在删除的时候被析构的：

```
Pool* p = new Pool(20);      //*p 现在被构造。
//...
delete p;                                  //*p 现在被析构。
```

如果对象既不是位于堆栈，又不是位于空闲存储区，那就只能是位于静态存储区。C++语言又是如何确定静态对象的构造和析构时刻呢？当编写这本书的时候，这个问题仍然处于 ANSI/ISO C++委员会的讨论之中。然而，我们相信 ANSI/ISO C++委员会将会做出如下几项保证：首先，当控制流程首次经过对象的定义时，局部静态对象应该被构造：

```
{   //一个模块的开始
    //...
    static Pool p(20);  //当控制流程首次经过这里时，p 被构造。
    //...
}
```

C++标准或许还会确保（除了一个特殊情况，我们将在后面讨论这个情况）：翻译单元中所有的非局部静态对象（非局部对象是指不在函数内部定义的对象）将会以在翻译单元中出现的顺序进行构造，并且以与出现顺序相反的顺序进行析构。例如，假设下面的代码出现在某个翻译单元的全局作用域中：

```
Pool p1(20);
//...
Pool p2(20);
//...
Pool p3(30);
```

那么 p1、p2、p3 将会以出现的顺序(即 p1、p2、p3)被构造，并以相反的顺序（即 p3、p2、p1）被析构。

我们上面所讲的（这种对非局部静态对象的构造和析构顺序的）特殊情况就是简单对象的构造和析构顺序。对于任何一个对象 X，如果下面两个条件都成立，我们就把这个对象 X

称为简单对象：

- x 不是一个类对象，或者 x 的类没有显式的构造函数。
- x 的定义把 x 初始化为一个编译期或者链接期常量。

C++语言可能也会确保在翻译单元 T 中定义的所有简单对象、非局部对象和静态对象，都会在 T 的对象代码被加载的时候（如果 T 是动态链接的，那么这可能要发生在程序开始执行的时刻之后）被构造。例如，在下面的翻译单元中：

```
Pool p1(20);
//...
int i = 7;  //一个简单对象的定义。
//...
Pool p2(20);
```

简单对象 i 的构造时刻将会是这个翻译单元的目标代码被加载的时刻。而这个时机发生在 p1 和 p2 被构造之前，也发生在翻译单元中的任何函数被执行之前。

关于静态对象的构造和析构时刻，ANSI/ISO C++可能还会增加额外的约定。然而，C++语言却未能给出这样的约定：非局部对象的构造发生在调用 main 函数的时刻之前。这个约定将会和对象文件的动态链接发生冲突。

C++程序员比较希望的一个约定就是在任何成功创建的程序中，对象总是在使用之前就构造好了，并在析构之后，不再使用这个对象。遗憾的是，C++语言并不能给出这样的约定。因为当程序被创建的时候，语言本身不总是可以决定是否存在一个构造非局部对象的顺序，据此对象所有的使用都会只位于对象的构造和析构之间（见练习 12.5）。

12.1.2　程序库的蕴含意义

如果 C++程序库定义了任何非简单的、非局部的静态对象，那么程序库用户需要创建的程序就有可能会在这个对象被构造之前，就使用这个对象（除非程序库实现者预先禁止这种使用，见 12.1.6 小节）。而这往往会使程序具有不确定的行为。例如，假设我们的程序库提供了下面的 Widget 类：

Widget.h:

```
class Widget {
public:
    void* operator new(size_t) {
        return pool.alloc();
    }
    void operator delete(void* p) {
        pool.free(p);
    }
    //...
private:
    static Pool pool;
};
```

```
Widget.c:
    Pool Widget::pool(sizeof(Widget));
```

我们使用了 Pool 来实现特定类的操作符 new 和 delete。并且 Widget::pool 是一个非简单的、非局部的静态对象。

现在假设下面的代码会出现在全局作用域中：

```
Widget* w = new Widget;
```

注意，w 也是一个非简单的、非局部的静态对象，并且 w 的定义和 Widget::pool 的定义将会包含在不同的翻译单元里。于是，当这个包含这些代码的程序被创建时，Widget::pool 可能会（也可能不会）在 w 被初始化之前就已经构造完毕。如果不会，那么 Widget::pool 将会在构造前就已经被使用（在 Widget::operator new 中）了。因此，这个程序将会有不确定的行为出现。

依赖编译系统在这些产生的代码内部插入检查也是可以的，这样任何未初始化对象的使用将会导致运行期错误。然而，这样做之后的结果是程序将会运行得非常地慢。另一方面，某些程序员希望能有一种可以选择检查与否的编译系统，以便在程序开发过程中，可以打开这种检查；而在效率受到巨大影响时，把这种检查关闭。但遗憾的是，现在并没有这种编译系统。

如果程序库包含了允许用户程序使用未初始化程序库对象的代码，那么要让用户辨识这个问题（指未初始化对象的使用）将会是困难的；用户即使辨识出了这个问题，也是无能为力的。

程序库应该避免这种问题（指用户程序使用未初始化的程序库对象）发生的可能性。通常说来，C++程序库就不应该定义和使用非简单的、非局部静态对象。当我们确实需要这类对象时，我们可以把这类对象移到空闲存储区来定义。例如，把 Widget::pool 移到空闲存储区将得到下面的代码：

```
Widget.h:
    class Widget {
    public:
        void operator new(size_t) {
            assert(pool != 0);
            return pool->alloc();
        }
        voic operator delete(void* p) {
            assert(pool != 0);
            pool->free(p);
        }
        //...
    private:
        static Pool* pool;      //这里是指针。
    };
Widget.c:
    Pool* Widget::pool = 0;
```

现在 Widget::pool 就是一个简单的、非局部的静态对象，因此可以约定这个对象将会在 Widget.c 的代码被加载的时候初始化，也就是说，在 Widget::operator new 执行之前已经初始化完毕。

既然我们把对象移到了空闲存储区，就必须在代码的某个位置分配和释放这些对象的内存空间，我们将在下面的几个小节讨论 C++程序库在这方面经常使用的几种做法。在讨论当中，我们将使用下面的 Widget 成员函数：

```
class Widget {
private:
    static void new_pool() {
        if(pool == 0)
            pool = new Pool(sizeof(Widget));
    }
    static void delete_pool() {
        delete pool;
    }
    //...
};
```

对于我们将要讨论的各种做法，没有一种是完美无缺的。因此，C++程序库设计者应该选择对特定程序库而言缺点最少的做法。

12.1.3　初始化函数

C++程序库处理静态初始化的方法之一就是提供一个程序库初始化函数。例如，包含类 Widget 的程序库可以提供一个全局函数 initwidgetlib，每个程序在使用该程序库的功能之前，都要先调用这个函数：

```
void initwidgetlib() {
    Widget::new_pool();
    //...
}
```

(另外，这个函数必须是 Widget 的友元。因此，在调用程序库的这个初始化函数之前，使用程序库的任何功能都应该在文档中记录为具有不确定的行为)

提供了一个初始化函数的程序库必须提供一个结束化函数，当程序使用这个程序库完毕时，将调用这个函数：

```
void finalizewidgetlib() {
    Widget::delete_pool();
    //...
}
```

(这个函数也必须是 Widget 的友元函数。)

然而，程序库的初始化函数具有几个不足之处。首先，要让程序库用户决定何时调用给定的初始化函数，有时是比较困难的。比较幼稚的策略是在 main 函数的开头就调用初始化函数：

```
int main() {
    initwidgetlib();
    initsomeotherlib();
    //...
}
```

为了使这种做法可以起作用，用户必须能够获得 main 函数的实现；然而这个要求有时候会阻碍这种依赖于初始化函数的实现。不仅如此，如果程序库的某些功能，在调用 main 函数之前，就已经用于某些静态对象的初始化，那么这个程序就会出现不确定的行为。因此，初始化函数实际上并没有解决静态初始化问题。

而且，即使程序并没有在静态对象的初始化阶段使用程序库，因此可以安全地等到执行 main 函数时才调用程序库初始化函数，但是要确定应该调用哪些程序库初始化函数也是困难的。假设有一个大型程序是由许多开发者组成的开发小组编写的。于是，main 函数的开发者就很难确定程序中究竟直接或者间接地调用了哪些程序库。即使 main 函数的开发者最后能够确定程序应该调用哪些程序库初始化函数，但在决定程序库初始化函数的调用顺序时，仍然会出现很大的问题。例如，假设程序库 X 的某些部分使用了程序库 Y 的某些部分，那么程序库 Y 的初始化函数调用就必须出现在程序库 X 的初始化函数调用之前。可以想象，在大的项目中记住这些依赖性简直就是一场噩梦。

不仅如此，程序库初始化函数还有另外一个问题：它有时候会初始化多于程序库需要的内容。假设某个程序只使用了 Widget 程序库的某些部分，并且这些使用没有涉及到 Widget 的类。然而，当用户调用 initwidgetlib 时，程序将会分配内存给 Widget::pool，而程序可能并不需要使用 Widget::pool，于是就浪费了内存空间。与其只提供一个单一的初始化函数，程序库还不如包含多个初始化函数，每个函数只初始化已经经过文档化的程序库子集。然而，这种解决方式却很快就会使程序变得难以控制。

最后，程序库初始化函数也不能很好地和模板结合。假设 Widget 是一个类模板：

```
template<class T>
class Widget {
    //...
private:
    static Pool* pool;
};
```

当程序使用了 Widget<T>对象时，initwidgetlib 就必须为所有的类型 TWidget<T>::pool 分配内存空间；然而遗憾的是，对于编写代码而言，这种实现是很困难的，要么就是不可能实现的。

12.1.4 初始化检查

程序库处理静态初始化问题的另一种方法是使用初始化检查。使用这个方法，当需要使用任何一个空闲存储区内的对象时，我们必须先确认这个对象的内存空间已经分配成功：

```
class Widget {
```

```
public:
    void* operator new(size_t) {
        new_pool();
        //现在我们可以安全地使用 pool 了。
        //...
    }
    void operator delete(void*) {
        new_pool();
        //现在我们可以安全地使用 pool 了。
        //...
    }
    //...
};
```

然而，初始化检查拥有两个缺点。首先，它会导致用户的程序变慢。幸运的是，通过对程序库代码的流程分析，初始化检查的次数可以大大减少。例如，如果存在某个正确使用 Widget 类的程序，那么只有对那些使用 Widget::operator new 构造的 Widget 对象，才会调用 Widget::operator delete。因此，我们就可以安全地删除 operator delete 中的初始化检查。

如果给定的初始化检查不能被删除，那么可以把它们移到调用次数比较少的位置。例如，假设我们的程序库包含了下面的函数：

```
Widget* create_widgets(int n) {
    while(--n > 0) {
        Widget* w = rew Widget;
        //...
    }
}
```

上面的代码中，每次循环都会调用初始化检查。假设针对我们程序库的实现，类 Widget 是私有类；并且不会直接调用 Widget::operator new 函数，而是只在 create_widgets 里调用 operator new。那么，我们就不应该在 Widget::operator new 里面执行初始化检查，而是把这个检查移到 create_widget 函数里面：

```
Widget* create_widget(int n) {
    Widget::new_pool();      //把初始化检查移到这里。
    while(--n > 0) {
        Widget* w = new Widget;
        //...
    }
}
```

现在就只在每次调用 create_widget 函数时，才会调用一次初始化检查。（为了实现这个改变，我们还必须把 create_widget 函数声明为 Widget 的友元。）

那些希望根据程序库的流程分析，来删除或者转移初始化检查的程序库实现者，应该为将来的开发者和代码维护者提供每个优化的详细文档。另外，对程序库实现的改变有时也会使原来的流程分析不再有效。

初始化检查的另一个缺点是这个技术并不能帮助我们决定何时删除对象。在我们这个例子里，Widget::pool 指向的对象应该在不再需要的时候被删除；而且，决定删除的时机是非常重要的。实际上，某些程序库就只使用初始化检查，而从没有删除分配的对象。例如，如果我们知道我们的用户并不会在意是否从没有删除 Widget::pool 对象，那么我们就可能不会花费精力来删除这个对象。（这种方法违反了我们在 4.5.2 小节的必须尽快释放资源的建议。）

12.1.5 初始化对象

程序库处理静态初始化的第三个方法就是使用初始化对象。在下面，我们将展示如何对我们的 Widget 例子使用初始化对象。首先，我们先把如下定义放到 *Widget.h* 中：

```
Widget.h:
    class Widgetinit {
        static int counter;
    public:
        Widgetinit();
        ~Widgetinit();
    };
    class Widget{
        friend class Widgetinit;
        //...
    };
    static Widgetinit widgetinit;        //初始化对象。
```

在这里，widgetinit 就是初始化对象。我们还把类 Widgetinit 声明为 Widget 的友元类。我们将如下实现 Widgetinit：

```
Widget.c:
    int Widgetinit::counter = 0;
    Widgetinit::Widgetinit() {
        if(counter++ == 0)
            Widget::new_pool();
    }
    Widgetinit::~Widgetinit() {
        if(counter-- ==1)
            Widget::delete_pool();
    }
```

(在一个实际程序库里，初始化对象将会初始化和结束化 Widget 需要的所有对象，而不仅仅是 Widget::pool)

让我们做一个合理的假设，对于任何确实使用 Widget（即不只是声明 Widget 类的名称）的用户，如果是第一次包含 Widget.h 头文件，那么在 Widget::pool 指向的 Pool 对象被分配之前，程序是不可能使用这个对象的。我们之所以能够确信这种不可能性，是因为在同一个翻译单元中，初始化对象 widgetinit 的初始化时刻，将会比任何与 Widget 对象有关的非简单对象的初始化时刻早；并且，如果 Widget::pool 对象还没有被初始化，那么 widgetinit 的初始化代码将会初始化这个 Widget::pool 对象。

然而，初始化对象也有几个缺点。首先，它们会意外地增加用户程序的代码量。例如，假设 Widget.h 包含了标准 iostream 头文件 iostream.h，并且假设，和许多 iostream 的实现一样，我们用户包含的 iostream 使用了一个初始化对象。那么每个使用 Widget 的程序——即使这个程序没有使用 iostream 的功能——将会引入初始化 iostream 程序库所需要的所有代码。为了避免这种多余的代码引入，我们可以在需要的时候直接声明 iostream 类，而不是包含 iostream.h 头文件（见 4.2.1 小节）。

相似地，假设下面的代码出现在我们程序库的实现里：

```
Widget.c:
    #include<iostream.h>
    void func1() {
        //...
        cout << "Make it so!";
        //...
    }
    void fun2() {
        //...
    }
```

函数 func1 使用了 iostream 程序库，但函数 func2 并没有使用这个程序库。当使用了依赖于初始化对象的 iostream 实现时，任何直接或者间接调用 func2 的用户程序，也会引入 iostream 库的初始化代码。为了避免这类代码引入，我们可以把 func2 函数移到另一个文件，而这个文件并不包含 iostream.h 头文件。

初始化对象的第二个缺点是可能会增加用户程序的运行时间。假设某个程序在许多翻译单元都包含了 Widget.h（Widget.h 也可以包含在某个其他的头文件里，而这个头文件本身被项目中的许多文件所包含，这样也算是包含了 Widget.h）。那么每个翻译单元都会包含 Widget 的初始化对象（即 widgetinit）的定义，而这些初始化对象可能会被零零散散地分发到整个程序的目标代码中；因此在虚拟内存系统中，当程序启动的时候，调用所有的初始化对象的构造函数，将会导致所有包含初始化对象的页在内存中发生故障（可能是频繁换页等），从而导致明显的延迟。

初始化对象的第三个缺点是，和初始化函数一样，它不支持模板。我们将把对这些细节的考虑留给读者（见练习 12.1）。

12.1.6　双构造

有许多技术（例如双构造）允许 C++程序库定义和使用非简单、非局部的静态对象，用户程序在初始化该对象之前使用它的可能性也很小。例如，假设类 Pool 有一个特殊的构造函数，它肯定不执行任何代码：

```
class Pool {
public:
    enum Do_nothing_ctor { do_nothing_ctor };
```

```
    Pool(Do_nothing_ctor) { /*executes no code */ }
    //...
};
```

枚举 Do_nothing_ctor 只是用来为提供这个特殊的构造函数服务的——通过重载。程序库实现者还必须确认被隐式调用的基类和成员变量的构造函数都不执行任何操作。

有了上面这个特殊的构造函数，我们就可以不需要再把非简单、非局部的静态对象 Widget::pool 移到空闲存储区了，我们可以重新改写 new_pool 函数（这个函数第一次出现在 12.1.2 小节）。如下所示：

```
#include<new.h>
class Widget {
private:
    static void new_pool() {
        static bool first_time = true;
        if(first_time) {
            new(&pool) Pool(sizeof(Widget));
            first_time = false;
        }
    }
    static Pool pool;
    //...
}
```

在这里，我们使用了操作符 new 的 placement(定位)版本。我们现在就可以使用从 12.1.3 小节到 12.1.5 小节讨论的各种技术，来确保 Widget::pool 是在初始化之后才使用的。注意，双构造并没有排除使用初始化函数、初始化对象或者初始化检查的必要性，但是它允许程序库使用非简单的、非局部的静态对象(从而使程序库不需要把这些对象都转移到空闲存储区)。

任何使用 Widget 的程序都需要构造 Widget::pool 对象两次：一次在第一次调用 new_pool 的地方，另一次在 Widget::pool 的定义位置（即最后的 static Pool pool 处）。为了使定义位置的构造函数调用不产生副作用（如前所述，我们不能确定这类调用发生的顺序——而这正是问题的关键所在），我们让这个构造函数不执行任何代码：

```
Pool Widget::pool(Pool::do_nothing_ctor);
```

你可能会认为双构造技术很不优雅。而某些 C++实现会结合使用双构造和初始化对象，以实现 iostream 程序库。但是，我们建议不在任何其他的程序库使用双构造技术。如 Plauger[Pla95]解释的那样，这项技术并不能排除在程序中使用未初始化的双构造对象的可能性。

12.2 局部化开销原则

可重用代码的一个很重要的设计目标就是局部化开销。对于一个系统，如果只花费了所使用特性的开销（即不付出其他多余的开销），那么我们就称这个系统符合局部化开销原则。

日常生活中到处都有局部化开销原则：一个按实物计价的色拉吧会使用局部化开销——顾客只需支付他所用食物的价钱；而纽约圣诞节前夕某个俱乐部的露天吧就不会使用局部化开销了——即使那些不喝饮料的顾客，也要为其他人喝的饮料付钱（通过入场费）。有时，开销是被故意分派的：当我们购买医疗保险的时候，我们都同意支付小部分的开销，而保险公司再把这些钱分发给少数受害者。

局部化开销原则几乎可以应用于计算机科学的所有领域。例如，在硬件领域，精简指令集计算机（RISC）的目标就是在机器指令集中重新确定局部化开销。随着时间的推移，指令集变得越来越复杂，从而每个程序都会为某些高度特化、不经常使用的指令的存在而花费大量开销——也就使程序执行变得更慢。然而，RISC 机器的设计目标成为可以在更精简的指令集合中更快地处理每条指令；于是，通过减少许多不常用的指令，局部化开销又回到了硬件设计的范畴。

12.2.1　局部化开销和 C++

C++许多特性的设计目标是使特性的实现局部化开销。就是说，程序员不应该为他们不使用的特性而花费开销——包括编译时间、链接时间、运行时间、代码大小或者增加语言的复杂性。然而，没有语言可以很好地满足这么多限制。可以想象，每在参考手册中添加一些内容，就会使编译器变得庞大一些，也会给标准委员会留下多一些讨论的内容。然而，语言的设计者必须决定这些被提议的特性的用处是否可以抵消它所带来的开销。

让我们考虑 C++的虚函数和局部化开销。显然，虚函数会带来下列开销：

- 在 C++语言的大多数实现中，与非虚函数调用相比，每个虚函数调用只是多需要 2 个到 5 个机器指令。
- 在特定的实现环境中，大多数现今的实现都不能内联虚函数调用（见 4.4.2 节）。
- 大多数实现都会使用（虚函数）表来决定应该调用哪个虚函数实例。于是，在包含有一个或多个虚函数的类的每个对象中，都会存在一个指向虚函数表的指针，而且，这些表和指针还会占用（用户编译后代码的）内存空间。
- 由于在类对象中指向虚函数表的指针的存在，使这个对象布局和 C 语言不再链接兼容。

除了增加了语言的复杂性之外，上面的每个开销都是局部化的——就是说，只有那些包含具有虚函数类的程序才会涉及到这些开销。而且，执行速度也是严格局部化的：只有当调用虚函数时，那些使用虚函数的程序才会花费调用开销。

12.2.2　局部化开销和程序库

当设计和实现 C++程序库的时候，要违反局部化开销原则是很容易的。程序库设计者应该知道时刻都有违反局部化开销原则的可能，并要尽量避免违反这种原则。如我们在 12.1.5 小节讨论的为了初始化 iostream 库的初始化对象，而导致的代码引入就是很好的例子；再重

复一遍，如果没有非常仔细地设计和实现程序库，就很容易违反局部开销原则。

现在考虑另一个例子，下面是某个程序库中的类：

```
class T {
protected:
    inline void f() {
        //...
    }
    //...
};
```

为了加快调用 f 函数的代码的执行速度，我们内联了 f 函数的定义。然而，程序库的一些用户要求需要从类 T 派生，并且改写 f 函数。因此，在程序库的下一个版本中，程序库提供者可能会把 f 函数声明为虚函数，如下所示：

```
class T {
protected:
    virtual inline void f() {
        //...
    }
    //...
};
```

遗憾的是，这个改变将会违反局部化开销原则。如在 4.4.2 小节讨论的那样，当编译器不能轻易地确定调用虚函数的对象类型时，现今的大部分 C++实现都不能内联虚函数调用。因此，如果要继续保持 f 为虚函数，f 的调用者就不能改写 f 函数；否则（即不实现为虚函数）就必须增加外联调用 f 函数的执行时间，并且也可能会由于某些程序能够改写 f 而增加 f 的外联拷贝（见 4.3.2 节）代码的大小。

另一方面，不把 f 声明为虚函数也可能会被认为是违反了局部开销化。于是，如果从那些希望可以更快地调用 f 函数的用户的角度考虑，所有的用户都应该放弃改写 f 函数的能力（见 4.4.2 小节，它讨论了何时应该内联某个函数，何时应该把这个函数声明为虚函数）。

12.3 内生类和外生类

在这一节里，我们将澄清一个经常被程序库设计者混淆的问题：内生容器和外生容器的区别。假设 List<T>描述链表，链表元素值的类型为 T。于是，我们可以用两种方法来实现 List。内生实现将直接把值存储在底层链表节点内部，如下面图 12.1 所示：

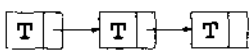


图 12.1 List 的内生实现

而外生实现将会把包含的值存储在单独的对象里面，如图 12.2 所示：

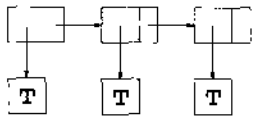


图 12.2　List 的外生实现

从上两图可以看出，内生容器和外生容器的区别只在于实现，而并不在于接口。如果不考虑如何实现的话，那么一个 List<T>应该只表现为一个链表，链表元素值的类型为 T。例如，如下的相等函数：

```
template<class T>
bool operator==(const List<T>& l1, const List<T>& l2);
```

应该在 l1 和 l2 描述相等的链表（指 l1 和 l2 具有相同个数的元素，并且相对应元素都是相等）时，将返回真值，而与使用上面哪种实现无关。例如，如果 List 是外生的，那么图 12.3 给出的 List<int>对象 l1、l2 和 l3 应该都是相等的。然而，如果 l1、l2 和 l3 都是内生的、元素类型为 T*的链表（这个指针类型结合图 12.3 和内生的定义就可以看出来），那么 l1 等于 l2，但它们都不等于 l3（因为指针指向的地址不同）。

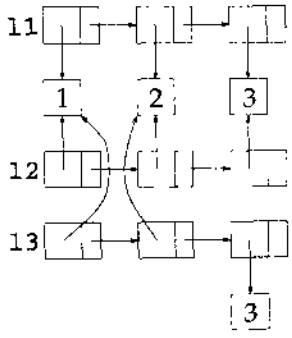


图 12.3　List 对象的相等性。如果 List 是外生的，那么 l1、l2 和 l3 相互之间都是相等的；如果 List 是内生的，那么 l1 等于 l2，但都不等于 l3

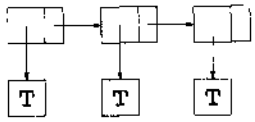


图 12.4　元素类型为 T 的外生链表或者元素类型为 T*的内生链表

注意，元素类型为 T 的外生链表和元素类型为 T*的内生链表的相同之处和不同之处，就如图 12.4 给出的那样，这两个链表的图看起来是一模一样。然而，它们的行为又是迥然不同的。

许多程序库设计者经常犯的一个错误就是提供一个类，对某些操作，这个类模拟元素类型为 T*的内生容器，而对其他操作，该类模拟元素类型为 T 的外生容器。考虑下面的类和相应的函数：

```
template<class T>
class Ptrlist {
public:
    void insert(T* t);
    //...
};
template<class T>
bool operator==(const Ptrlist<T>& l, const Ptrlist<T>& m);
```

假设这个类的文档如下：

Plist<T>是元素类型为 T*(指向 T 的指针)的链表。函数 insert 把 t 插入到表头。当比较 l 和 m 时，函数 operator==将会提领（defrenrence）每个包含在链表 l 和 m 中的指针。

除了第一个句子以外，Plist 的设计者好像并不没有决定 Plist<T>究竟是要模拟元素类型为 T*的链表，还是元素类型为 T 的链表；其中，抽象性的定义和 insert 函数的定义表明的是前者（即类型 T*），而 operator==的定义却意味着后者（即类型 T）。可以看出，如果 Plist 确实模拟的是元素类型为 T*的链表，那么 operator==将不具备正规语义（关于正规语义见 2.2 节）。

某些读者可能会认为用户通常都会让指针容器具有非正规语义。然而，对于希望提供非正规类的程序库设计者，他们必须清楚而且详细的文档化每个非正规类，才能避免往用户代码中不知不觉地引入错误，因为用户通常都认为所有的类都是正规类，而且这种认为也是非常合理的。

12.4 迭代器

迭代器通常都会和容器类一起使用，而且定义和实现他们的语义也是相当重要的，因此，很值得我们在这一节里花费笔墨进行重点阐述。假设我们的程序库提供了一个 Map 类（见 3.2.2 小节）：

```
template<class X, class Y>
class Map {
public:
    void insert(const X& x, const Y& y);
    void remove(const X& x);
    //...
};
```

在上面的代码中，函数 insert 在删除了所有首值为 x 的映射对后，将会把<x, y>对添加到所给映射中；函数 remove 将删除所有首值为 x 的映射对。（还有其他用来设计 Map 类的方法，请参考练习 4.4。）

Map 的用户可能需要一种可以迭代映射中所有值的方法。通常，给容器用户提供的最好迭代方法就是为该类提供一个伴随的迭代器类。下面就是 Map 类的迭代器类：

```
template<class X, class Y>
class Map_iter {
```

```
public:
    Map_iter(const Map<X,Y>& m);
    bool next(X& x, Y& y);
    //...
};
```

Map_iter 的构造函数构造了一个对象 m 的迭代器。只要 m 中还含有没有被迭代器返回的映射对，函数 next 就会把某个剩余的映射对分别赋值给 x 和 y，并返回真值；当没有剩余的映射对时，next 函数就返回假值。但是，映射对返回的顺序是未经指定的。下面就是 Map_iter 的一个典型用法：

```
Map_iter<X,Y> i(m);
X x;
Y y;
while(i.next(x,y)) {
    //(x,y)是 m 中的下一个映射对。
    //...
}
```

你可能已经注意到，我们对 Map_iter 行为的定义是不完全的。例如，下面的代码，将会有什么样的行为呢？

```
Map_iter<X,Y> i(m);
X x;
Y y;
while(i.next(x,y)) {
    m.insert(some_x, some_y);        //行 1
    m.remove(another_x);             //行 2
    //...
}
```

这里，用户迭代 m 中所有的映射对，在迭代过程中，插入新的映射对，并删除旧的映射对。那么，当迭代的时候，插入和删除的语义是什么呢？在行 1 的插入操作之后，迭代器 i 是否会返回新插入的映射对呢？如果执行行 2 的时候，i 还未能返回 another_x，那么将会有什么行为呢？

对于这些有问题的操作，最简单的处理办法就是把它们看作非法操作。例如，我们可以给 Map::insert 和 Map::remove 强加前提条件：对于给定的映射，不存在外部的迭代器。（违反这些前提条件的后果会是 5.3 节所描述的可能结果之一。）禁止这些程序库的操作并不会给用户对程序库的理解造成困难，而且实现起来也非常容易和高效。然而，禁止后函数的语义并不是许多用户所希望得到的，因为在迭代的时候，改变容器往往是很有用的操作。

与其对这些有问题的操作进行禁止，还不如文档化这些未经指定的操作，如在对 Map 对象的迭代过程中，我们根本就不能确定：在这个迭代器中，插入和删除的值最后是否是可见的，对此我们可以提供这个未经指定操作的文档。然而，这个做法也是不可取的，因为与完全指定行为的接口相比，一个未经指定的接口是更加容易导致错误的（见练习 12.13）。而且，至少有些用户只愿意编写依赖于指定行为的代码。

对于在迭代过程中希望具有指定行为的程序员，通常都希望 insert 和 remove 具有如下的语义：

1. 除了无效的迭代器（应该是指向 last 的迭代器）之外，添加到容器的值应该对所有的迭代器都是可见的。

2. 对于原先未能看见某些值的迭代器，如果删除这些值，那么这些迭代器也是看不见这些值的。

遗憾的是，要高效地实现这些行为是非常困难的

假设我们用二分查找树来实现 Map(见 3.2.1 节)。树中的每个节点存储一个映射对，并使用二分查找树的插入算法来插入映射对，二分查找树的删除算法来删除映射对。显然，Map_iter 的实现需要遍历树中的节点。假设最近调用 next 而返回的映射对所对应的节点记为节点 a，该实现还应该把从根节点到节点 a 的路径保存在迭代器的内部状态里面。现在考虑下面的代码：

```
m.insert(x,v);
```

假设当执行这个 insert 调用的时候，x 并不在 m 的定义域里面，从而，二分查找树插入算法就会把一个包含值 x 的节点插入到 m 的树中。现在假设刚插入值 x 的这个树节点已经被 m 的迭代器 i 遍历过了，那么如果 i 继续遍历这棵树的话，它将不会再次迭代这个刚插入 x 值的节点，也就不会看到这个节点。而这种行为违反了上面部分 1 的语义要求。

我们可以让每个 Map_iter 对象都对 Map 树进行多次遍历，并标记哪些节点已经早先遍历过了，而且当最后一次遍历表明已经没有新的节点存在时，就结束遍历，从而来解决上面的问题。然而，这个实现将会使代码的运行速度非常地慢。因为，每次对 Map 树的迭代，至少都需要进行两次，并且当在迭代的过程中插入一个新的节点，就必须对整棵树都再迭代一遍。

一个用于实现上面语义的比较高效的方法就是为 Map 的实现增加一个线性链表。如果存在任何原有 Map 外部的迭代器（即不是对原先存在于树中的节点进行操作，如插入等），Map::insert 将会把这个给定的值插入到这个链表中，而不是直接把含这个值的节点插入到树中。当迭代器完成了整棵二分查找树的迭代之后，迭代器将继续迭代这个链表。另一方面，删除节点操作可以在被删除节点作上标记。当最后一个对 Map 的原有迭代器被删除（即不再有插入或者删除操作存在）以后，才把链表中的值合并到二分查找树中去，并且把作上标记的节点删除。关于有这些 Map 和 Map_iter 的操作而导致的效率问题，我们将让读者自己来分析。

12.5 类耦合

对两个类来说，如果其中类的接口或者实现使用了另外一个类，我们就说这两个类是耦合的。考虑一个描述 C++解析器的类 Parser。因为解析器需要操纵许多字符串，因此，如果使用 String 类，而不是直接操纵 char*类型的值，将会使 Parser 的实现更加简单。这样 Parse 和 String 就是互相耦合的。相似地，如果我们需要在 Parser 实现中使用哈希表，我们可能就

会使用类 Hashtable 来实现类 Parser，而不是直接使用内建的数组和指针类型来编写代码。

某些程序员认为一般情况下耦合是不可取的，而其他程序员则认为耦合通常都是一个很好的主意。实际上，耦合是既有优点又有缺点的。如我们的 Parser 例子所示，耦合可以简化程序库的实现，耦合还可以简化程序库的使用。譬如，假设 Parser 的构造函数需要接收一个目录链表，通过这个目录链表，来查找需要包含的文件。下面的代码将让 Parser 类和两个类（List 和 String）相耦合：

```
class Parser {
public:
    Parser(const List<String>& directories, /*...*/);
    //...
};
```

这里，List 是用于描述链表的某个类。而如果我们不被允许耦合这 List、String 或者其他类的话，我们应该如何设计 Parser 的构造函数呢？我们可能需要用户给 Parser 传递一个指向指针数组的指针，指针数组的元素是指向以 null 结尾的字符串的指针。代码如下：

```
class Parse {
public:
    Parser(const char* const* directories, /*...*/);
    //...
};
```

然而，这个接口使用起来非常困难，而且与使用 String 和 List 的接口相比，这个接口显然是很容易导致错误的。并且，与构造一个元素为 String 的 List 相比，构造一个元素为指向以 null 结尾的字符串的指针的、以 null 指针结束的数组将会要求更多的代码。不仅如此，用户可能会忘记需要用一个 null 指针来结束这个数组。

另一方面，耦合也可能会使程序库的使用变得困难。假设我们希望 Parser 的用户指定某个宏定义的初始化集合。例如，我们希望能够允许用户创建一个 Parser 对象，这个对象是以解析了下面的代码之后解析器的状态来创建的：

```
#define MACRO1 STUFF
#define NULL_MACRO
```

接下来，我们可以使用 12.4 节的 Map 类来提供所期望的 Paser 类的功能：

```
class Parser {
public:
    Parser(const List<String>& directories,
        const Map<String,String>& initial_defines,
        /*...*/);
    //...
};
```

于是，用户就可以编写如下代码：

```
Map<String,String> initial_defines;
initial_defines.insert("MACRO1","STUFF");
initial_defines.insert("NULL_MACRO", "");
```

```
//...
Parser p(directories,initial_defines,/*...*/);
```

为了使用 Parser，用户必须先理解 String、List 和 Map；因此，希望使用 Parser 的程序员必须先学习这 3 个类。

另一种设计方法就是只使用两个类，代码如下：

```
class Parser {
public:
    Parser(const List<String>& directories,
        const List<String>& initial_defines,    /*...*/);
    //...
};
```

用户现在就可以如下编写代码：

```
List<String> initial_defines;
initial_defines.insert("MACRO=STUFF");
initial_defines.insert("NULL_MACRO=");
//...
Parser p(directories, initial_defines, /*...*/);
```

现在程序员如果想使用 Parser，就只需多学习两个类 List 和 String。因此，在这里，耦合的类越少，使用就越容易。

为了完全理解一个类，我们必须先理解在这个类接口中使用的所有其他的类，和在其他的类的接口中使用的所有另外的类，并以此类推。如果程序库是高度耦合的，那么用户可能会不愿意为了理解所有耦合类之间的关系而深入钻研程序库的文档。

对任何效率的测量，耦合都可以产生正面的影响和负面的影响。例如，为了能在 C++的解析器中提高使用效率，Hashtable 已经高度优化过了，那么解除 Parser 和 Hashtable 之间的耦合可能就会增加运行时间。另一方面，考虑在 Parser 接口中使用 List，这个耦合需要用户首先编译 List 的定义，而且还可能会把有关 List 实现的代码引入到用户程序中。但是使用指针数组的 Parser 版本就不会有这两项开销。因此，就运行速度而言，List 版本可能会比解耦合版本慢；因为与在指针数组中迭代所有的值相比，在 List 中迭代所有的值的速度将要花费更多的时间。

耦合的另一个缺点是就提取（见 1.1.1 节）而言，耦合类比非耦合类更加困难。例如，如果 Parser 使用了 String、Hashtable、List 和 Map，那么任何希望在我们的程序库中提取 Parser 的用户，都会花费很多时间来判断 Parser 究竟直接或者间接地使用了哪些类，并且在提取的时候，需要把这些被使用的类一起提取。

对于所建议的耦合，只有优点比缺点多时（也只有在这种情况下），程序库的设计者才应该进行耦合。

12.6 推迟决定

程序库的设计者通常不会具备足够的信息来作出某个特殊的设计决定。于是，这个设计

者可以推迟这个决定，并让用户参加到做决定这个过程中来。例如，假设我们在编写用于网络计算环境的程序——网络计算环境是指在这个环境里面，不同的机器连接在一起，并且可以互相通信。并且假设我们希望为我们的用户提供一个 location 函数，它接收一个机器名称作为参数，并返回这个机器的位置。location 的返回类型是 Loc，Loc 是我们程序库的一个类：

```
Loc location(const String& machine_name);
```

location 的这个实现决定于这个机制：在用户的网络系统中，在一台机器上执行的程序，可以得到有关另一台机器的信息（远程过程调用就是一种实现方案）。对于某些网络系统，调用 location 可能会是很慢的。譬如，如果调用 location 的程序的所在机器位于德国的慕尼黑，而 machine_name 对应的机器位于澳大利亚，那么完成 location 的调用肯定需要比较长的时间。由于这个原因，让我们开始寻求加快 location 调用的方法吧。

一种普遍用于加快函数执行速度的技术就是缓存结果。当使用一个给定的参数集合调用缓存函数时，这个函数先检查它是否已经以这些参数被调用过了；如果没有，它将利用参数计算原函数的结果，并在函数返回之前，把这个结果存储高速缓存中；如果有（即函数已经以这些参数调用过了），那么它就只返回高速缓存中的结果。下面是函数 f 的缓存版本的伪代码，它只接收一个 int 类型参数：

```
int memoized_f(int n) {
    if(this function has already been call with the value n,
        return the value cached for n
    int result = computation of f(n);
    store result in this function's cache
    return result
}
```

如果 f 真是一个参数为 n 的函数（并且它的返回值并不依赖于全局变量的值），那么缓存 f 就一定会保持它原来的语义。如果我们经常使用相同的参数来调用 f，那么缓存 f 的结果有时就可以大大提高函数的运行速度。

假设我们经常使用相同的参数值来调用 location。然而，如果机器的位置是经常变动的，那么 location 也不能完全由参数 machine_name 来决定。幸运的是，机器的位置是很少变动的。于是，进一步假设我们的用户希望一个更快的 location——虽然有时候（但很少）这个函数会出错，而一个慢的 location 肯定不会出错。

为了缓存 location，我们需要实现一个 cache(高速缓存)，因为 cache 是一个从 String 到 Loc 的映射，所以我们可以使用 3.2.2 小节到 3.2.4 小节的 Map 类：

```
Loc location(const String& machine_name) {
    static Map<String,Loc> cache;
    if(cache.contains(machine_name))
        return cache.valueat(machine_name);
    Loc answer;
        //进行网络连接，并得到答案（即位置），
    cache.insert(machine_name,answer);
    return answer;
```

```
}
```

(Map 函数 valueat 将会返回映射中第 1 个值（即参数值）所对应的第 2 个值。)

然而，假设在 Map 类的使用中，已经给予了我们的用户充分的灵活性。譬如，假设我们程序库中的 Map 类是一个抽象类，它有两个派生类 Small_map 和 Large_map；其中 Small_map 储存的映射对较少，而它的效率比 Large_map 的效率要高。于是，我们在上面声明的 cache 是非法的（因为 Map 现在已经是抽象类了），我们必须选择 Small_map 和 Large_map（或者其他的类）其中的一个。

假设由于 location 参数的原因，我们需要在 location 中使用一个更加高效的 cache。如果 location 只是在数量较少的机器之间相互调用，那么我们将使用 Small_map；否则我们就不得不使用 Large_map。遗憾的是，我们可能并不知道究竟有多少机器需要调用 location，因此我们只能从用户那边得到某种提示（或者帮助）以做出这个决定。

一种普遍用于获得程序库用户帮助的技术是提供一个模板，其中通过一两个模板参数来得到用户的帮助信息，从而根据这些信息做出决定。针对我们的问题，我们可以把 location 实现为模板函数，唯一的模板参数允许用户指定 cache 映射的类型：

```
template<template Catch<class X,class Y> >
Loc location(const String& machine_name);
```

(这里，我们使用了一个 C++新特性：模板参数本身也可以是一个模板。)另外，这个模板的实现和 location 非模板版本的实现是很相似的：

```
template<template Catch<class X, class Y> >
Loc location(const String& machine_name) {
    static Catche<String, Loc> catch;
    //...
}
```

现在用户可以编写如下代码：

```
Loc l1 = location<Small_map>(machine_name1);
```

或者：

```
Loc l2 = location<Large_map>(machine_name2);
```

如果 location 用于 Cache 的合法模板参数只有 Small_map 和 Large_map，那么程序库设计者就应该文档化这个模板参数限制条件（关于更多的文档化模板参数限制条件，见 11.4.4 小节）。

在作决定前获得用户的帮助也是有缺点的。首先，最后的程序库接口通常都会暴露程序库的实现细节。例如，我们的 location 模板就暴露了关于这个函数如何实现的某些细节。显然，暴露细节通常都是不明智的，也是应该尽量避免的。其次，获得用户的帮助会使程序库的使用更加困难。例如，编写代码

```
Loc l = location<Small_map>(machine_name);
```

就比不上编写下面的代码方便：

```
Loc l = location(machine_nanme);
```

通常，我们可以提供一个模板默认参数来改善这个问题：

```
template<template Catch = Small_map>
Loc location(const String& machine_name);
```

现在，用户如果并不在意 cache 使用的映射类型，就可以如下编写代码：

```
Loc l = location<>(machine_name);
```

然而，与原来的（非模板）接口相比，这个接口仍然比较难以使用。而且，声明默认模板参数的能力是 C++新增的特性之一。因此，依赖于默认模板参数的代码有时会是不可移植的。

12.7 总结

C++程序库的设计者需要知道静态初始化问题。我们建议程序库不要定义和使用任何非简单的、非局部的静态对象。如果确实有必要实现这些对象，就应该在空闲存储区实现这些对象。为了给这些对象分配内存和进行初始化，程序库可以使用初始化函数、初始化检查和初始化对象，其中每种方法都有各自的缺点。

对于任何程序库设计，局部化开销都是一个很重要的考虑因素。一个程序如果没有使用程序库的某个特性，它就不应该蒙受和这个特性存在相关的开销。

容器是一种很重要的可重用类。容器类的设计者应该细心地挑选容器类的设计类型：内生的或者外生的，但绝对不能是两者的混合体。用正确的语义来设计迭代器也非常需要谨慎和注意。

通常情况下，在程序库中类的耦合可以简化程序库的实现。耦合有时还可以使程序库的使用变得容易。而在某些时候，耦合也可以使程序库的使用变得困难。因此，对于是否要使用耦合，程序库设计者应该认真地衡量耦合带来的优点和缺点。

有时，可以通过把决定推迟给用户来避免做出左右为难的决定。一个普遍用于推迟决定的技术就是让用户指定一个模板参数来提供帮助信息。

12.8 练习

12.1　我们在 12.1.5 小节声明说初始化对象不能用来解决类模板的静态初始化问题。假设重新改写 12.1.5 小节开头处的 Widgetinit 类，让它成为一个类模板。说明这个类模板不能用于初始化 12.1.4 小节开头的 Widget 类模板。

12.2　考虑 12.1.5 小节中类 Widgetinit 的析构函数，为什么当 Widget::counter 的值为 1 时，这个析构函数要调用 Widget::delete_pool 呢？请给出解释。

12.3　考虑 12.1.2 小节末尾处 Widget::pool 的定义，为什么我们不能把这个定义改成如下代码呢？请给出解释：

```
Pool* Widget::pool = new Pool(sizeof(Widget));
```

12.4　某些人曾经提出一种处理静态初始化问题的办法，就是为那些在静态初始化过程

中不能被调用的程序库函数提供文档。

a. 在静态初始化过程中，位于 12.1.2 小节开头的 Widget 类的哪些成员函数是不能被调用的？

b. 程序库中不直接依赖于非简单的、非局部的静态对象初始化的函数，可以调用另一个依赖于这类静态对象的函数。于是，为了能够文档化在静态初始化期间，不能被调用的函数，请给出一个算法，用来识别这类不能被调用的函数。

c. 当使用这个办法来处理静态初始化问题时，将会有什么样的兼容性问题发生？

d. 当使用这个办法来处理静态初始化问题时，最大的缺点是什么？

e. 与这一章里表述的另外 3 个方法（即初始化对象、初始化函数和初始化检查）相比，你如何评价这个方法？

12.5 （**）对于 C++程序 P 中的非简单的、非局部的静态对象，我们定义一个它的线性初始化顺序。例如，对于程序 P 中的初始化顺序 o，如果无论如何执行程序 P，都不会导致有未初始化的对象被使用，我们就称 o 是可以接受的。现在考虑下面的函数：

```
                    ┌─ true   如果 P 具备一个可接受的初始化顺序
orderable(P) ──<
                    └─ false  如果 P 不具备一个可接受的初始化顺序
```

a. 证明函数 orderable 并不是可计算的（即不一定存在）。

b. 对于 a 部分在 C++编译系统编译后的结果，讨论这个结果的隐含意义。

12.6 （*）对于这一章里给出的用于解决静态初始化对象的 3 个办法——初始化对象、初始化函数和初始化检查，讨论它们 3 者之间的关系。其中哪个违反局部化开销原则的程度最严重（见 12.2 节）？哪个最好地符合局部化开销原则？请在执行速度和占用空间这两个方面考虑上面两个问题。

12.7 （*）借助于 C++中的虚基类，你可以编写如下的类体系：

```
class Link { /*...*/ };
class Sublist1 : virtual public Link { /*...*/ };
class Sublist2 : virtual public Link { /*...*/ };
class List : public Sublist1,
             public Sublist2 { /*...*/ };
```

Sublist1、Sublist2 和 List 对象都有一个（也只有一个）它们公共基类 Link 的实例。因此，对 Sublist1 对象、Sublist2 对象和 List 对象而言，Link 类分别相对于 Sublist1、Sublist2 和 List 的占用位置是不一样的（这些占用位置是指内存中基类和派生类的排列位置，可以不需要理解）。（如果你对这一点不清楚，在所有的 3 个派生类对象中，你可以试着画出一个 Sublist1、Sublist2 和 List 的对象布局，在结构中，Link 相对于 Sublist1 和 Sublist2 的位置是相同的；于是，就可以看出这样的结构就是我们应该尽力避免的钻石继承结构。）

虚基类通常都是通过间接实现的。每个拥有虚基类的对象都有一个指针（存储在与对象的开头位置有固定偏移量的位置），这个指针指向基类子对象。而且，所有对基类成员的存取都是间接的。

a. 针对执行速度而言，这个虚继承的实现满足局部化开销原则吗？

b. 针对空间使用而言，这个虚继承的实现满足局部化开销原则吗？（为了回答这个问题，应该考虑当这个程序把某个对象存在内存时，这个对象就是否处于使用状态。）

12.8　假设我们尝试通过提供两个版本的成员函数：一个是非虚函数，另一个是虚函数，但这个虚函数的功能只是简单地调用这个非虚函数，来解决 12.2.2 小节所描述的虚函数和内联之间的冲突，这个设计的缺点是什么？

12.9　在多重继承被添加到 C++之前，许多 C++编译器为虚函数调用

```
p->f();
```

产生的代码大概如下：

```
(p->vptr[index])(p);
```

这个代码提领（dereference）p 来查找 p 类型对象的虚函数表，然后表利用索引来找到需要调用的函数，接下来用 p 指向的对象来调用这个函数：

当多重继承添加到 C++之后，即使在不使用多重继承的程序中调用和上面相同的函数的代码将变成：

```
(p->vptr[index])(p+p->vptr[index].delta);
```

这个函数用和上面相同的方法来查找需要调用的函数，但是在它可以调用之前，为了应用 p 的值，它会查找一个 delta——因此传递给这个函数的参数会是相应类型对象的地址。（如果你还没有完成练习 12.7，那么先解决练习 12.7，因为它有助于理解为什么对 p 应用一个 delta 是很有必要的。）

a. 针对运行时间而言，这个实现违反局部化开销原则吗？如果是，是如何违反的？

b. 针对内存使用而言，这个实现违反局部化开销原则吗？如果是，是如何违反的？

12.10　考虑下面一个异常处理机制的实现：每个函数的目标代码中都包含有异常处理代码，当这个函数被调用的时候，首先判断异常是否存在，如果异常存在，那么目标代码中的异常处理代码将会在堆栈中把这个函数调用删除，就像这个函数从没有被调用一样。

a. 这种实现是违反局部化开销原则的，请解释原因。

b. 假设用户可以这样告诉编译器：无论异常存在与否，我这个函数是不能被从堆栈中删除的。那么这样的实现符合局部化开销原则吗？

c. b 部分实现的异常处理机制的优点和缺点各是什么？

12.11　如果为包含许多类和函数的程序库只提供一个头文件，这样会违反局部化开销原则吗？如果会，那么是如何违反的？

12.12　在 12.4 节讨论的 Map_iter 类中，为了返回（借助于引用参数）迭代中的下一个映射对，每次调用 Map_iter::next 都必须至少执行一次 X 的赋值操作和 Y 的赋值操作。

a. 为何我们不优化这个赋值操作，在迭代过程中，让它直接返回 m 内部存储的下一个映射对拷贝的引用，其中 m 是正被迭代的 Map 对象。

b. 我们可以这样优化 next：让 next 返回一个指向 m 中内部存储的下一个映射对的拷贝

的指针（而不是一个引用）。这个优化对于安全性、使用难易程度和 Map_iter 的实现都有哪些影响呢？如果你是 Map_iter 的设计者，你会应用这种优化吗？

12.13 假设我们这样定义 12.4 节 Map_iter 类的语义：在对 Map 的迭代过程中，我们并没有指定所添加或者删除的值最后是否可见。给出一个例子，在这个例子中，用户如果完全指定迭代器的语义，并且使用该类的接口，那么用户就会犯下一个本来可以避免的错误。

12.14 假设你在开发练习 1.2 和练习 7.1 所讨论的 Path 类。考虑下面的函数，它使用了你程序库的另一个类 List：

```
class Path {
public:
    List<Path> expand_wildcards() const;
    //...
}
```

函数 expand_wildcards 将会查找程序所在机器的文件系统，并且返匹配（根据某些定义好的规则集合）给定 Path 的文件列表。那么，上面 Path 和 List 的耦合有些什么样的优缺点呢？

12.15 练习 2.9 提到了克林法则（对于任何一个语言，如果（也只有当）这个语言可以表达为正则表达式，这个语言才能被有限数量的接受者所接受）。考虑练习 2.9 的 a 部分，假设你的程序库提供了一个 FSA 类来模拟有限数量的接受者，还提供了一个 Regex 类来模拟正则表达式。接下来，为了模拟克林法则，你要给出所提供的类和函数。对于你在练习 2.9 中 a 部分的解决方案中所创建的类，请给出耦合这些类的两个缺点。

12.16 在 12.6 节，我们讨论了两种版本的 location：第 1 种版本运行速度比较慢，但绝对是正确的；第 2 种版本，也就是缓存版本，运行比较快，但有时会出错。给出设计一个包含程序库的方法，让用户可以选择他们所希望的行为。

12.17 （*）给出一个例子，这个例子说明了使用 Map(包含实现细节)来设计 location 是不明智的。

12.9 参考文献和相关资料

Rgiser[Rei92]和 Stroustrup[Str94a]的 3.11.4 小节都讨论了静态初始化问题。Buroff 和 Murray[BM94]讨论了在静态初始化问题面前，实现全局常量对象的困难。Plauger[Pla95]进一步深入地讨论了双构造。

设计一个 C++容器类和可以在这个容器迭代对象的机制是非常困难的，而且还没有很好的解决办法；Koenig[Koe92a,Koe92b]讨论了相关的一些问题。现今的 C++程序库（例如，标准组件库[UNI92]、全国健康协会程序库[GOP90]、Tool.h++[Rog92]、Booch 组件程序库[Rat91]和 GNU C++程序库[Fre88]）都使用了各种容器和迭代器类型。Koenig[Koe93b]讨论了由于容器类和真值（truth value）之间的联系而引起的设计问题。Tarjan[Tar93]讨论了内生和外生容

器。

12.4 节所描述的二分查找树和线性链表数据结构之间的结合，是由 Steve Buroff 提出，并引起我们的注意的。

Carroll[Car93]进一步讨论了耦合。Weinand、Gamma 和 Marty[WGM89]给出了一个设计得很好的 C++程序库的例子，在这个库中，存在许多类的耦合。

Springer 和 Friedman[SF89]的第 11 章很好地讨论了缓存技术。

中英文术语对照表

A

abort, handling errors by exit or 借助程序退出或者程序中止来处理错误
abstract 抽象（的）
    base class 基类
    base class 也见接口类
    const C++关键字
abstraction 抽象
    documenting class 文档化抽象类
    documenting in terms of 基于抽象情来编写文档
access protection and name lookup, order of 存取保护检查和名字查找，两者的顺序
acquisition 获得
    is initialization technique, resource 在初始化时获得资源
    of externally reused library 获得外部可重用的程序库
added example added 函数例子
adding member function, source compatibility of 增加成员函数、增加成员函数的源代码兼容性

alignment restrictions, portability and 赋值约束　可移植性和赋值约束
alternative version of new and delete 另一种 new 和 delete 版本
Annotated C++ Reference Manual . See ARM 一本由 BJ 和 Ellis 合著的 C++查询手册
ANSI/ISO

B

build	创建（生成）
build process, portability of, sequence	创建过程，创建过程的可移植性，创建过程的顺序
build-time efficiency	生成期的效率
bzero example	bzero 例子

C

C run-time library	C 运行期程序库
C++	
ANSI/ISO definition of	ANSI/ISO C++的定义
extensions, nonportability and nonstandard	扩展性、不可移植性和非标准的 C++
implementation details, nonportability and	不可移植性和 C++的实现细节
language definition changing	C++语言定义的改变
language definition, nonportability caused by changing	由于 C++语言定义的改变而导致的不可移植性
run-time library	C++运行期程序库
casting	强制转型
away const,	去除 const 的强制转型
in pointer container, safe	在指针容器中的安全转型
char, implementation-defined signedness of	char(字符型)定义实现的符号
class	类
interface class	接口类
documenting class abstraction	文档化类的抽象
class conversions required by special-purpose	特殊目的要求的类转型
definition of exception safety of class	类的异常安全性定义
definition of nice class	nice 类的定义
class definition	类的定义
one-definition rule for class	类的一处定义原则
depth of class	类的深度
documenting syntactic interface of class	文档化类的语法接口
fanout of class	类的深度
nice class	nice 类
root	根

concrete class	具体类
definition of concrete class	具体类的定义
conflict	冲突
conflict created by external reuse	重用外部代码所导致的冲突
definition of conflict	冲突的定义
environmental name conflict	环境名称冲突
global name conflict	全局名称冲突
head file name conflict	头文件名称冲突
macro name conflict	宏名称冲突
prevention conflict via namespace construct	通过名字空间来避免冲突
conformant code	构造代码
nonportablility of conformant code	构造代码的不可移植性
consistency	一致性
consistent state	一致的状态
const	C++关键字
abstract const	抽象 const
bit const	位 const
casting away const	去 const 的强制转型
maximal use of const	最大化使用 const
type hole created by reinterpretation of const	重新解释 const 所导致的类型漏洞
construction	构造
constructor	构造函数
copy constructor	拷贝构造函数
default constructor	默认构造函数
when called constructor	当调用构造函数的时候
const and nonrefrence parameter	const 和非引用参数
container	容器
pointer container	指针容器
inheritance-based design of container class	基于继承的容器类设计
endogenous vs. exogenous	内生容器和外生容器
context of reusable code	可重用代码的上下文（环境）
control flow	控制流程
conventions	约定
name convertions	命名约定

D

inheritance semantics	文档化继承的语义
lifetime of returned reference	文档化返回引用的生存期
private member	文档化私有成员
regular function	文档化正规函数
syntactic interface of class	文档化类的语法接口
template argument restriction	文档化模板参数的限制
template instantiation transitive dependencies	文档化模板实例化传递的依赖性
double construction	双构造
downcast	向下转型
virtual derivation and downcast	虚派生和向下转型
downgrading	细分
dynamic	动态的
linking and run compatibility	动态链接和运行兼容性
linking and static initialization	动态链接和静态初始化
link library	动态链接库
link library versioning mechanism	动态链接库的版本机制

E

ease	容易性
ease of implementation	实现容易性
effective code size	有效代码大小
efficiency	效率
and ease of implementation, portability vs.	效率与（实现容易性和移植性）的关系
and reusability	效率和重用性
build-time	创建期效率
compile-time	编译期效率
of automatic template instantiation	模板自动实例化的效率
of coupling	耦合的效率
of external reuse	重用外部代码的效率
of init checks	初始化检查的效率
of init object	初始化对象的效率
of multiple inheritance	多重继承的效率
of self-contained library	自含式程序库的效率

F

G

H

handle class	句柄类
documenting,	文档化句柄类
link compatibility provided by,	句柄类提供的链接兼容性
handling errors	处理错误
by correcting problem,	通过纠正问题来处理错误
by creating nil value,	通过创建一个 nil 值来处理错误
by exit or abort,	通过程序退出或中止来处理错误
by interpreting invalid data as valid,	通过将无效数据看作有效数据来处理错误
by returning error value,	通过返回错误值来处理错误
by throwing exception,	通过抛出异常来处理错误
head file conflict,	头文件冲突
heap,	堆
hierarchy	体系
history of reuse	重用的历史
hoisting	提升
hoisting member fuction of template	提升模板成员函数
hole	漏洞
type hole	类型漏洞

I

implementation	实现
implementation-defined	实现定义的
implicit conversions	隐式转型
imcompatibility	不兼容性
inconsistent state	不一致状态
independence	独立性
template parameter independence	模板参数的独立性
inheritability	继承性
obstacles,	继承性的障碍
increased by moving code into virtual function	通过把代码移入虚函数来提高继承性
of interface class	接口类的继承性
inheritance	继承

interface	接口
minimal standard interface	最小标准接口
consistency	接口一致性
of class	类的接口
interfaced inheritance hierarchy	接口继承体系
interface class	接口类
derived assignment problem solved with	通过接口类而解决的派生赋值问题
inheritability of,	接口类的继承性
link compatibility provided by	接口类提供的链接兼容性
virtual derivation of	接口类的虚派生
internet	互联网
invalid data as valid, handling errors by interpreting	通过把无效的数据解释为有效的数据来处理错误
invalidated	使无效
invariant	不变性
broken by deep copy	由深拷贝而破坏的不变性
broken by shallow copy	由浅拷贝而破坏的不变性
representation,	描述不变性
invasive inheritance	入侵继承
iostream library	iostream 库
irregular	非正规的
irregular semantics	非正规语义
isolation caused by self-contained library	由于自含式程序库所导致的(与其他程序库)的分离
interation	迭代
interators	迭代器

K

kit	工具箱
Kleene's theorem	克林法则

L

layout	布局

of multiple inheritance	多重继承的局部化开销
of virtual function	虚函数的局部化开销
long	C++关键字，长整型

M

macro	宏
manual	人工的，手动的
manual template instantiation	人工模板实例化
directive	人工模板实例化指令
maximal use of const	最大化使用 const
member function	成员函数
as obstacle to inheritability	给继承性带来障碍的成员函数
documenting automatically generated	文档化自动生成的成员函数
source compatibility of adding	增加成员函数的源代码兼容性
memmozed version of function	函数的缓存版本
memory leaks resulting from inheritance-base design	基于继承设计所导致的内存泄漏
minimal standard interface	最小化标准接口
minimizing	最小化
free store via effective algorithms	通过使用高效算法来最小化空闲存储区的使用空间
init checks with control flow analysis,	用控制流程分析来最小化初始化检查的次数
multiple	多重
inheritance	多重继承
ownership of conversions	转型的多重所有权
mutable	C++关键字
myths	神话
of inlining	内联的神话
of reuse	重用的神话

N

name	名称，名字，命名
conflict	名称冲突

member function as obstacle to inheritability 被看作继承性障碍的非虚成员函数

O

object 对象
simple object 简单对象
object code 目标代码
portability of 目标代码的可移植性
object factory 对象工厂
link compatibility provided 由对象工厂所提供的链接兼容性
used to hide implementation decision, 用于隐藏实现决定的对象工厂
obstacle to reuse 重用的障碍
nontechnical, 重用的非技术性障碍
technical 重用的技术性障碍
obstacle to inheritability 继承性的障碍
excess data members as （被认为是继承性障碍的）成员变量存取
friend as （被认为是继承性障碍的）友元关系
inheritance-preventing member function as （被认为是继承性障碍的）不利于继承性的成员函数
nonvirtual derivation as （被认为是继承性障碍的）非虚拟派生
nonvirtual member function as （被认为是继承性障碍的）非虚拟成员函数
overprotection as （被认为是继承性障碍的）过度保护
undermodularization （被认为是继承性障碍的）模块化不足
one-define rule 一处定义原则
for class definitions, 类的一处定义原则
for function definitions 函数的一处定义原则
implications for library 程序库蕴含意义的一处定义原则
open file limit 打开文件数目的限制
optimization opportunity created by inlining 由内联所获得的优化机会
optimized variant of library 优化过的程序库变量
order 顺序
of access protection and name lookup 存取权限保护和名称查找的顺序
of argument evaluation, unspecified 未经指定的参数求值顺序

P

R

S

of adding member function	添加成员函数的源代码兼容性
space	(存储)空间
special-purpose	专用的
class	专用类
specialization	特化
stack, definition	堆栈的定义
stack overflow	堆栈溢出
stack space	堆栈存储空间
used by stack space	庞大对象所使用的堆栈存储空间
standard	标准的
interface	标准接口
minimal standard interface	最小标准接口
Standard Template Library	标准模板库
standard-conforming code, portability and,	可移植性和符合标准的代码
static	静态的
linking	静态链接
object, free store used by	释放静态对象所使用的存储空间
storage, definition	静态存储空间的定义
static initialization	静态初始化
dependencies, nonportability and	不可移植性和静态初始化的依赖性
dependencies, unspecified,	未经指定的静态初始化的依赖性
dynamic linking and	动态链接和静态初始化
problem	静态初始化问题
problem and init check	静态初始化问题和初始化检查
problem and init function	静态初始化问题和初始化函数
problem and init object	静态初始化问题和初始化对象
store	存储
struct overlaying, portability and,	移植性和结构重叠
substitutability, definition	多态的定义
SVID run-time library	SVID 运行库
switching type	可以转换的类型
symmetric conversions	对称转型
synchronization, external reuse and release	外部重用和版本同步
synonym mechanism for naming conventions	用于命名约定的同义机制

T

U

参考文献

[AHU83] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[ANS89] ANSI. American National Standard for Information Systems — Programming Language C. Technical Report X3.159–1989, ANSI, December 1989.

[ANS94] ANSI. Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++. Technical Report X3J16/94–0158, ANSI, September 1994.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AT&89] AT&T. *System V Interface Definition.* Addison-Wesley, third edition, 1989.

[Baa88] S. Baase. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, second edition, 1988.

[Bec94] P. Becker. Writing portable C++ code. *The C++ Report*, 6(7), September 1994.

[Ben82] J. Bentley. *Writing Efficient Programs.* Prentice Hall, 1982.

[Ber90] L. Berlin. When objects collide: Experiences with reusing multiple class hierarchies. In *ECOOP/OOPSLA '90 Proceedings*, pages 181–193. ACM, 1990.

[BI94] K. Baclawski and B. Indurkhya. The notion of inheritance in object-orient programming. *Communications of the ACM*, 37(9):118–119, 1994.

[BM94] S. Buroff and R. Murray. C++ oracle. *The C++ Report*, 6(9), November–December 1994.

[BN94] J. Barton and L. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.

[BV93] G. Booch and M. Vilot. Simplifying the Booch Components. *The C++ Report*, 5(5), June 1993.

[Car92a] T. Cargill. *Elements of C++ Programming Style*. Addison-Wesley, 1992.

[Car92b] M. Carroll. Invasive inheritance. *The C++ Report*, 4(8):34-42, October 1992.

[Car93] M. Carroll. Design of the USL Standard Components. *The C++ Report*, 5(5), June 1993.

[Car94] T. Cargill. Exception handling: A false sense of security. *The C++ Report*, 6(9), November–December 1994.

[Car95] M. Carroll. Tradeoffs of run-time parameterization. *The C++ Report*, 7(1), January 1995.

[CHS91] B. Cohen, D. Hahn, and N. Soiffer. Pragmatic issues in the implementation of flexible libraries for C++. In *Usenix C++ Conference Proceedings*, pages 193–202, April 1991.

[CL95] M. Cline and G. Lomow. *C++ FAQs: Frequently Asked Questions*. Addison-Wesley, 1995.

[Cli90] W. D. Clinger. How to read floating point numbers accurately. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990. SIGPLAN Notices, 25(6).

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[Cog90] J. Coggins. Design criteria for C++ class libraries. In *Usenix C++ Conference Proceedings*, pages 25–35, April 1990.

[Cop92] J. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[DR90] N. Dershowitz and E. Reingold. Calendrical calculations. *Software Practice and Experience*, 20(9):899–928, September 1990.

[DSS90] S. Dorward, R. Sethi, and J. Shopiro. Adding new code to a running C++ program. In *Usenix C++ Conference Proceedings*, pages 279–292, April 1990.

[Dup95] L. Dupré. *BUGS in Writing: A Guide to Debugging Your Prose*. Addison-Wesley, 1995.

[DW83] M. Davis and E. Weyuker. *Computability, Complexity, and Languages.* Academic Press, 1983.

[Ede92] D. R. Edelson. Smart pointers: They're smart, but they're not pointers. In *Usenix C++ Conference Proceedings*, pages 1–19, August 1992.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Faf94] D. Fafchamps. Organizational factors and reuse. *IEEE Software*, 11(5):31–41, September 1994.

[FN91] M. Fontana and M. Neath. Checked out and long overdue: Experiences in the design of a C++ class library. In *Usenix C++ Conference Proceedings*, pages 179–191, April 1991.

[Fre88] The Free Software Foundation, Cambridge MA. *The GNU C++ Library*, February 1988.

[GOP90] K. Gorlen, S. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming.* John Wiley and Sons, 1990.

[HO87] D. Halbert and P. O'Brien. Using type and inheritance in object-oriented programming. *IEEE Transactions on Software Engineering*, 4:71–79, 1987.

[HS91] S. Harbison and G. Steele. *C, A Reference Manual.* Prentice Hall, third edition, 1991.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[HW67] J. Hodges and M. Whitten. *Harbrace College Handbook.* Harcourt Brace &World, sixth edition, 1967.

[IBM] *The IBM Systems Journal.* 32(4), 1993.

[IEE90] IEEE. Information technology — portable operating system interface (POSIX) part 1: System appolication program interface (API) [C language]. Technical Report 1003.1-1990, IEEE, December 1990.

[JR92] P. Johnson and C. Rees. Reusability through fine-grain inheritance. *Software Practice and Experience*, 22(12):1049–1068, December 1992.

[KA94] S. Kendall and G. Allin. Sharing between translation units in C++ program databases. In *Usenix C++ Conference Proceedings*, pages 247–264, April 1994.

[Kef93] T. Keffer. The design and architecture of Tools.h++. *The C++ Report*, 5(5), June 1993.

[KL92] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *OOPSLA '92*, pages 435-451, 1992.

[Kni93] A. Knight. Copying. *The Smalltalk Report*, 2(5), February 1993.

[Knu89] D. Knuth. The errors of TEX. *Software Practice and Experience*, 19(7):607–685, July 1989.

[Koe91] A. Koenig. Library design is language design. *The Journal of Object-Oriented Programming*, June 1991.

[Koe92a] A. Koenig. Accessing C++ container elements. *The Journal of Object-Oriented Programming*, July 1992.

[Koe92b] A. Koenig. Designing a C++ container class. *The Journal of Object-Oriented Programming*, February 1992.

[Koe92c] A. Koenig. Space-efficient trees in C++. In *Usenix C++ Conference Proceedings*, August 1992.

[Koe93a] A. Koenig. Functions that return references. *The C++ Report*, 5(8), October 1993.

[Koe93b] A. Koenig. Truth and equality in C++. *The C++ Report*, 5(7), September 1993.

[Lap87] J. Lapin. *Portable C and UNIX System Programming.* Prentice Hall, 1987.

[Lea93] D. Lea. The GNU C++ library. *The C++ Report*, 5(5), June 1993.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* McGraw-Hill, 1986.

[Lim94] W. C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, September 1994.

[Lin92] M. Linton. Encapsulating a C++ library. In *Usenix C++ Conference Proceedings*, pages 57–66, August 1992.

[Lip91] S. Lippman. *C++ Primer.* Addison-Wesley, second edition, 1991.

[LP94] M. Linton and D. Pan. Interface translation and implementation filtering. In *Usenix C++ Conference Proceedings*, pages 227–236, April 1994.

[LS93] M. Lee and A. Stepanov. The science of C++ programming. Invited lecture, ANSI C++ meeting, San Jose, CA, November 1993.

[Mar91] B. Martin. The separation of interface and implementation in C++. In *Usenix C++ Conference Proceedings*, pages 51–64, April 1991.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[Mey92a] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[Mey92b] S. Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, 1992.

[Mey92c] S. Meyers. Using C++ effectively: Approaches to effectiveness. *The C++ Report*, 4(6), July–August 1992.

[Mey93a] N. Meyers. Memory management in C++ (part 1). *The C++ Report*, 5(6), July–August 1993.

[Mey93b] N. Meyers. Memory management in C++ (part 2). *The C++ Report*, 5(9), November–December 1993.

[Mey94a] S. Meyers. Code reuse, concrete classes, and inheritance. *The C++ Report*, 6(6), July–August 1994.

[Mey94b] S. Meyers. `operator=`: The readers fight back. *The C++ Report*, 6(9), November–December 1994.

[Mey94c] S. Meyers. Our friend, the assignment operator. *The C++ Report*, 6(4), May 1994.

[MM92] G. McCluskey and R. Murray. Template instantiation for C+−. *ACM SIGPLAN Notices*, 27(12):47- 56, December 1992.

[MS94] D. Musser and A. Stepanov. Algorithm-oriented generic libraries. *Software Practice and Experience*, 24(7):623–642, July 1994.

[Mur88] R. Murray. Building well-behaved type relationships in C++. In *Usenix C++ Conference Proceedings*, pages 19–30, October 1988.

[Mur93] R. Murray. *C++ Strategies and Tactics.* Addison-Wesley, 1993.

[NB94] L. Nackman and J. Barton. Base-class composition with multiple derivation and virtual bases. In *Usenix C++ Conference Proceedings*, pages 57–72, April 1994.

[Nov92] Novell, Inc. *C++ Language System Release 3.1 Manual, Selected Readings*, 1992.

[Pla93] P. Plauger. Reusability myths. *Computer Language*, 10(5):25–29, May 1993.

[Pla95] P. Plauger. *The Draft C++ Library.* Prentice Hall, 1995.

[Rat91] Rational. *The C++ Booch Components*, 1991.

[RC90] A. Riel and J. Carter. Towards a minimal public interface for C++ classes. *The C++ Insider*, 1(1), 1990.

[RDC93] E. Reingold, N. Dershowitz, and S. Clamen. Calendrical calculations, II: Three historical calendars. *Software Practice and Experience*, 23(4):383–404, April 1993.

[Rei92] J. Reiser. Static initializers: Reducing the value-added tax on programs. In *Usenix C++ Conference Proceedings*, August 1992.

[Rog92] Rogue Wave Software. *Tools.h++ Class Library*, 1992.

[Sch90] J. Schwarz. C++ is not an object-oriented language. *The C++ Journal*, Fall, 1990.

[SF89] G. Springer and D. Friedman. *Scheme and the Art of Programming*. The MIT Press, 1989.

[SM77] D. Stanat and D. McAllister. *Discrete Mathematics in Computer Science*. Prentice Hall, 1977.

[Spr93] R. Sprowl. Abbreviated objects. *The C++ Report*, 5(9), November–December 1993.

[Sta94] W. Staringer. Constructing applications from reusable components. *IEEE Software*, 11(5):61–68, September 1994.

[Ste92] W. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[Str93] B. Stroustrup. Library design using C++. *The C++ Report*, 5(5), June 1993.

[Str94a] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str94b] B. Stroustrup. Making a vector fit for a standard. *The C++ Report*, 6(8), October 1994.

[SW79] W. Strunk and E. White. *The Elements of Style*. Macmillan Publishing Co., 1979.

[SW90] G. Steele and J. L. White. How to print floating point numbers accurately. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990. SIGPLAN Notices, 25(6).

[Tar83] R. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[Tea93] S. Teale. *C++ IOStreams Handbook*. Addison-Wesley, 1993.

[Tra88] W. Tracz. Software reuse myths. *Software Engineering Notes (ACM SIGSOFT)*, 13(1):17–20, January 1988.

[UNI92] UNIX System Laboratories. *C++ Standard Components Programmer's Reference*, 1992.

[Vil94] M. Vilot. An introduction to the STL library. *The C++ Report*, 6(8), October 1994.

[Vil95] M. Vilot. The C++ standard library. *The C++ Report*, 7(2), February 1995.

[WBF91] C. Will, J. Baldo, and D. Fife. *Proceedings of the Workshop on Legal Issues in Software Reuse*. Technical report, Strategic Defense Initiative Organization, July 1991. Published by Institute for Defense Analyses, IDA Document D-1004.

[Weg93] I. Wegener. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, (118):81–98, 1993.

[WGM89] A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[Wik87] Å Wikström. *Functional Programming Using Standard ML*. Prentice Hall, 1987.

[WOS94] B. S. Weide, W. F. Ogden, and M. Sitaraman. Recasting algorithms to encourage reuse. *IEEE Software*, 11(5):80–88, September 1994.